US006249866B1

(12) **United States Patent**
Brundrett et al.

(10) **Patent No.: US 6,249,866 B1**
(45) **Date of Patent: Jun. 19, 2001**

(54) **ENCRYPTING FILE SYSTEM AND METHOD**

(75) Inventors: **Peter Brundrett**, Seattle; **Praerit Garg**; **Jianrong Gu**, both of Bellevue; **James W. Kelly, Jr.**, Redmond; **Keith S. Kaplan**, Bothell; **Robert P. Reichel**; **Brian Andrew**, both of Redmond; **Gary D. Kimura**, Kirkland; **Thomas J. Miller**, Bellevue, all of WA (US)

(73) Assignee: **Microsoft Corporation**, Redmond, WA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **08/931,774**

(22) Filed: **Sep. 16, 1997**

(51) **Int. Cl.**[7] **H04L 9/30**
(52) **U.S. Cl.** **713/165**; 380/286; 713/201
(58) **Field of Search** 380/21, 4, 25, 380/30, 286; 713/165, 166, 200–201

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,238,854	*	12/1980	Ehrsam et al.	713/135
4,386,234	*	5/1983	Ehrsam et al.	380/21
5,142,578	*	8/1992	Matyas et al.	380/280
5,224,166	*	6/1993	Hartmann, Jr.	380/50
5,319,705		6/1994	Halter .	
5,598,470	*	1/1997	Cooper et al.	380/4
5,870,468		2/1999	Harrison .	
5,870,477		2/1999	Sasaki .	

FOREIGN PATENT DOCUMENTS

0 681 233 A1 11/1995 (EP) .

OTHER PUBLICATIONS

Copy of Written Opinion in Corresponding PCT Application No. PCT/US98/19049.
Copy of International Search Report in Corresponding PCT Application No. PCT/US98/19049.

Denning, Dorothy E.; Branstad, Dennis K. "A Taxonomy for Key Escrow Encryption Systems" *Communications of the ACM* vol. 39, No. 3: 34–39. Mar. 1996.
Blaze, Matt. "A Cryptographic File System for Unix" *First ACM conference* Nov. 3–5, 1993.
Rivest, R. L.; Shamir, A.; Adleman, L. "A Method for Obtaining Kigital Signatures and Public Key Cryptosystems" *Communications of the ACM* vol. 21, No. 2: 120–126. Feb. 1978.
Cattaneo, G.; Persiano, G. "Design and Implementation of a Transparent Cryptographic File System for Unix" *Dep. Informatica ed Appl., Universita di Salerno, Baronissi(SA)—Italy.*
Blaze, Matt. "Key Management in an Encrypting File System" *1994 Summer USENIX* 27–35. Boston, MA Jun. 6–10, 1994.
Eastlake, et al. "RFC 1750: Randomness Recommendations for Security" *GlobeCom Network* 1–28 Dec. 1994.
Editors: Foot, G. H.; Sewell, R. F. "The Public Key Cryptography" vol. 1, No. 1. Jan. 1991.
Editors: Foot, G. H.; Sewell, R. F. "The Public Key" vol. 1, No. 2. Jun. 1991.

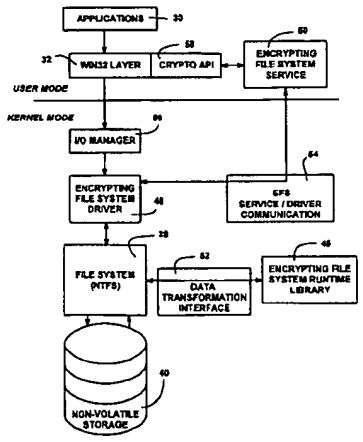
* cited by examiner

Primary Examiner—Gilberto Barron, Jr.
(74) *Attorney, Agent, or Firm*—Michalik & Wylie, PLLC

(57) **ABSTRACT**

A system and method for encryption and decryption of files. The system and method operate in conjunction with the file system to transparently encrypt and decrypt files in using a public key-private key pair encryption scheme. When a user puts a file in an encrypted directory or encrypts a file, all data writes to the disk for that file are encrypted with a random file encryption key generated from a random number and encrypted with the public key of a user and the public key of at least one recovery agent. The encrypted key information is stored with the file, whereby the user or a recovery agent can decrypt the file data using the private key thereof. When a proper private key is used, encrypted reads from the disk are decrypted transparently by the file system and returned to the user.

58 Claims, 20 Drawing Sheets



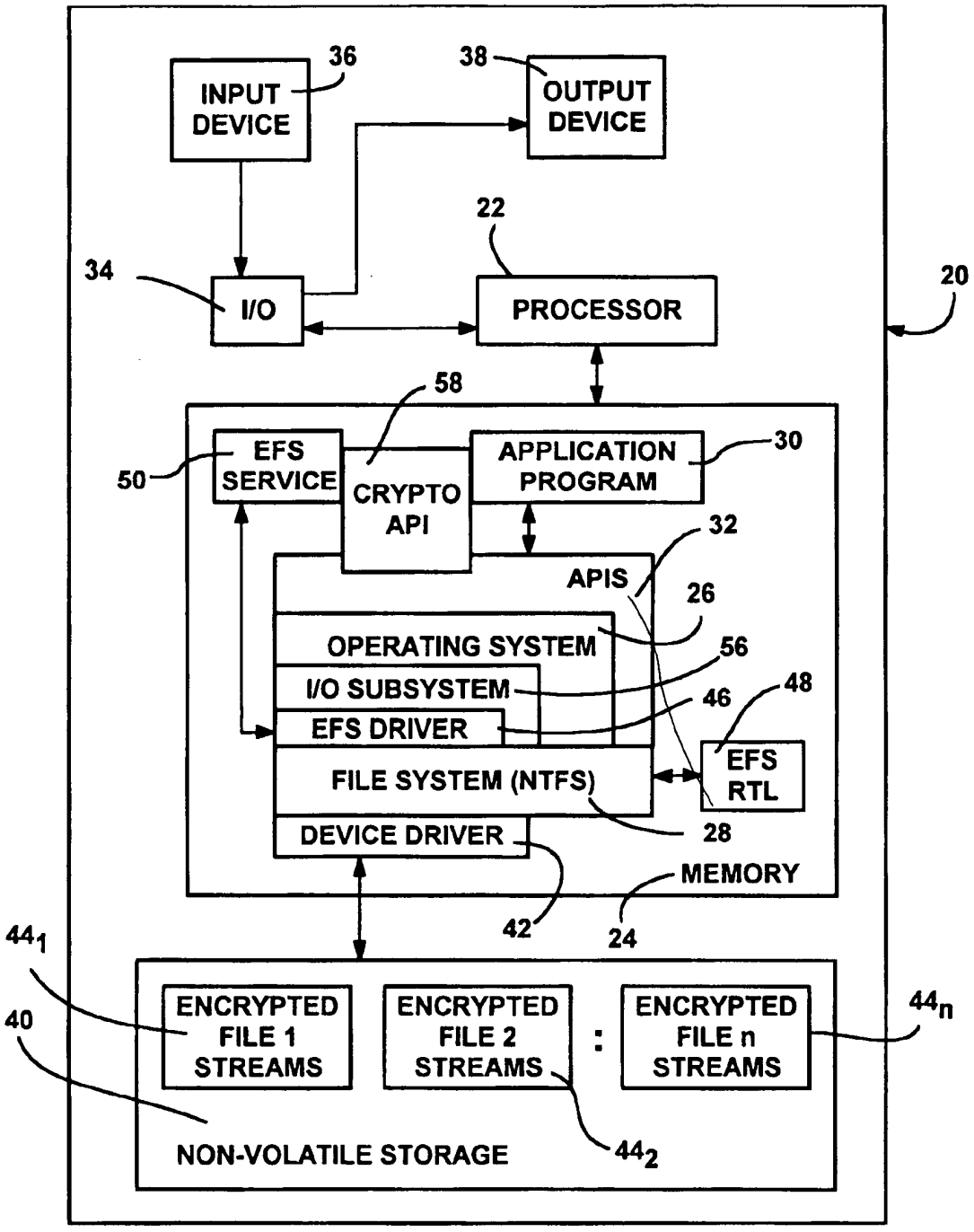

INPUT DEVICE
36
38
OUTPUT DEVICE
22
34
I/O
PROCESSOR
20
58
EFS SERVICE
50
CRYPTO API
APPLICATION PROGRAM
30
32
APIS
26
OPERATING SYSTEM
56
I/O SUBSYSTEM
46
48
EFS DRIVER
EFS RTL
FILE SYSTEM (NTFS)
28
DEVICE DRIVER
MEMORY
42
24
44_1
ENCRYPTED FILE 1 STREAMS
ENCRYPTED FILE 2 STREAMS
ENCRYPTED FILE n STREAMS
44_n
40
NON-VOLATILE STORAGE
44_2

FIG. 1

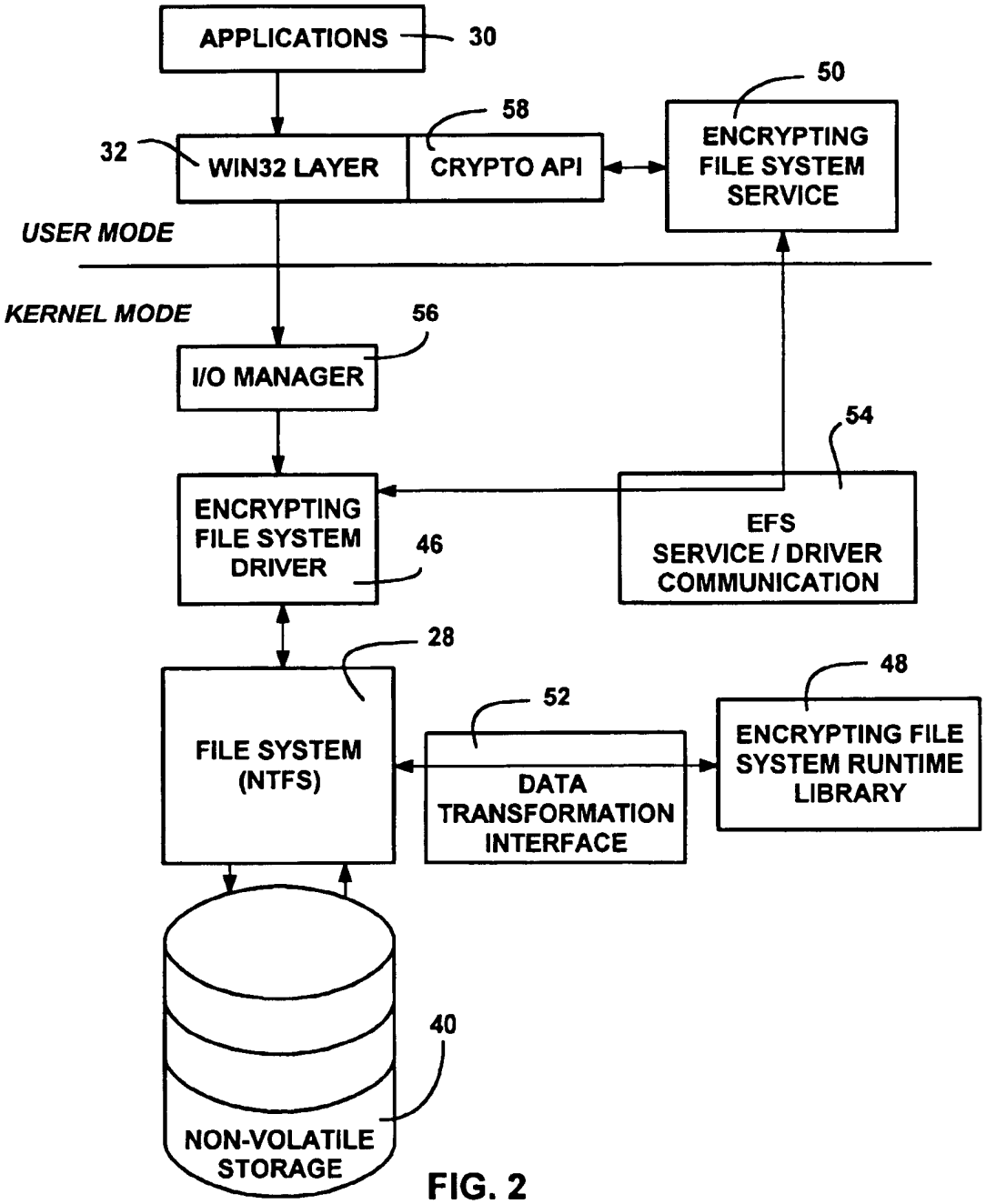

APPLICATIONS
30
50
58
32
WIN32 LAYER
CRYPTO API
ENCRYPTING FILE SYSTEM SERVICE
USER MODE
KERNEL MODE
56
I/O MANAGER
54
ENCRYPTING FILE SYSTEM DRIVER
46
EFS SERVICE / DRIVER COMMUNICATION
28
48
52
FILE SYSTEM (NTFS)
DATA TRANSFORMATION INTERFACE
ENCRYPTING FILE SYSTEM RUNTIME LIBRARY
40
NON-VOLATILE STORAGE

FIG. 2

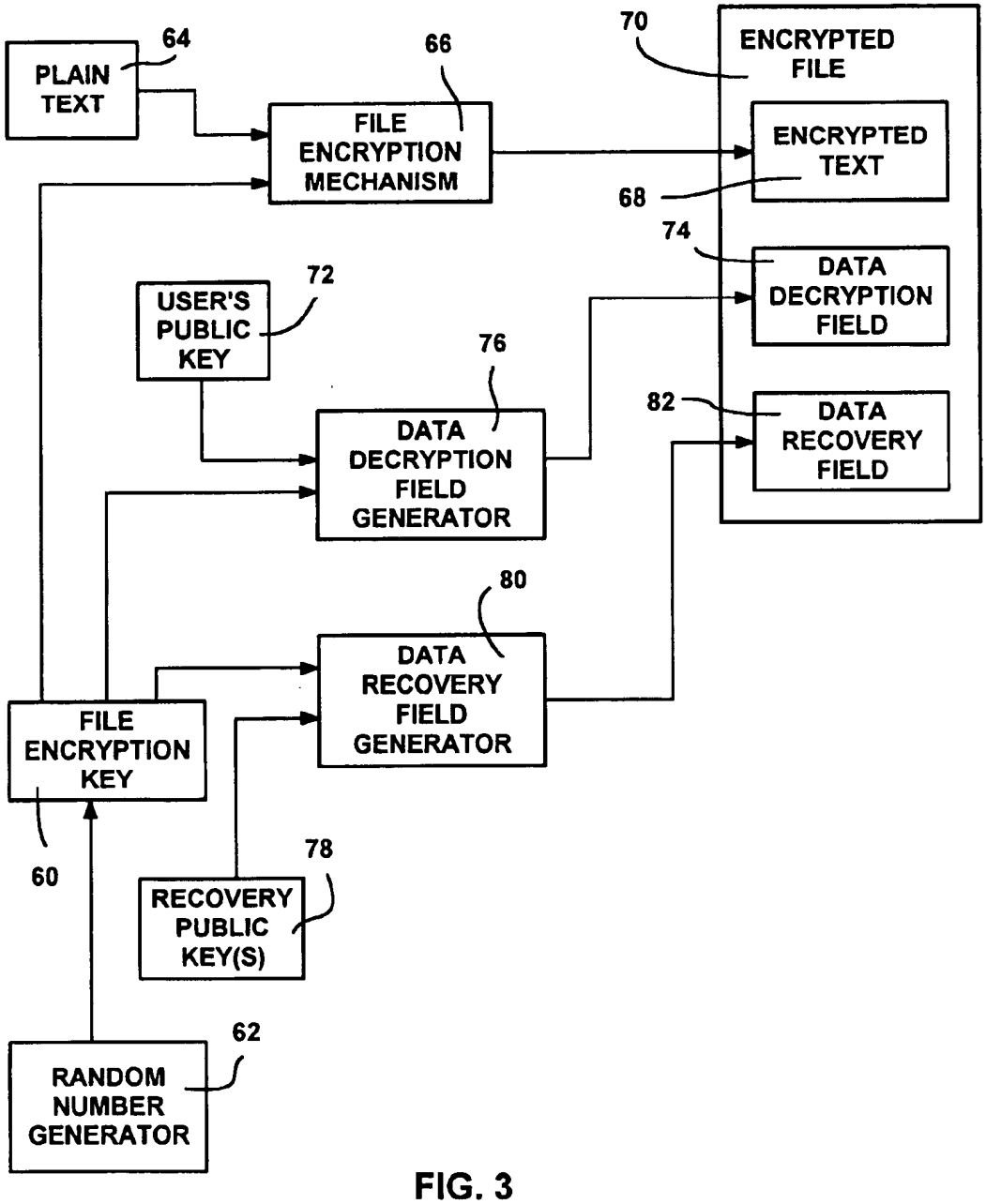

PLAIN TEXT
64
FILE ENCRYPTION MECHANISM
66
70
ENCRYPTED FILE
ENCRYPTED TEXT
68
74
DATA DECRYPTION FIELD
USER'S PUBLIC KEY
72
76
DATA DECRYPTION FIELD GENERATOR
82
DATA RECOVERY FIELD
80
DATA RECOVERY FIELD GENERATOR
FILE ENCRYPTION KEY
60
78
RECOVERY PUBLIC KEY(S)
62
RANDOM NUMBER GENERATOR

FIG. 3

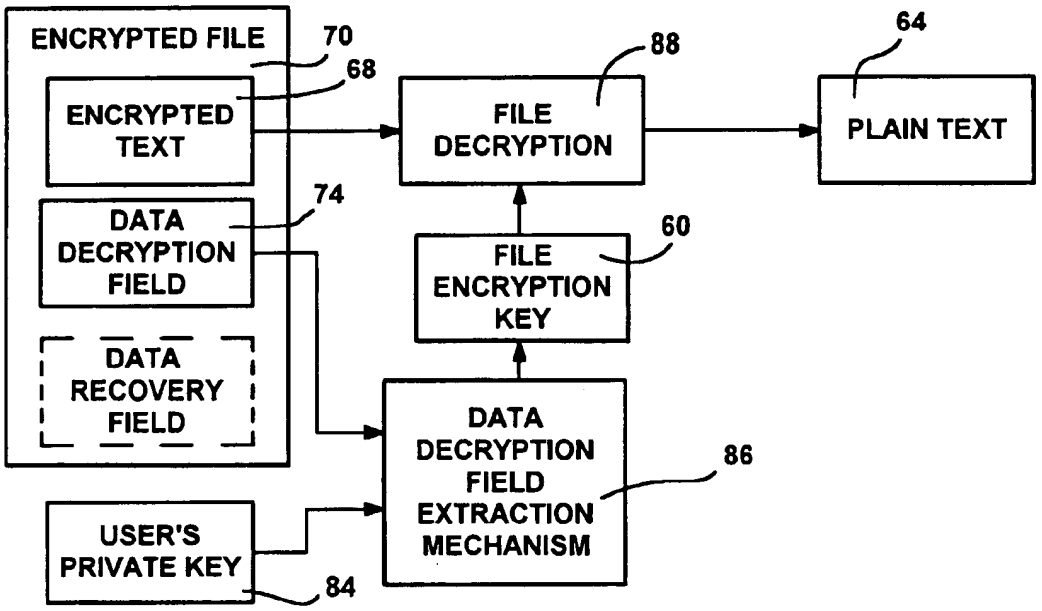

ENCRYPTED FILE
70
68
ENCRYPTED TEXT
DATA DECRYPTION FIELD
74
DATA RECOVERY FIELD
USER'S PRIVATE KEY
84
FILE DECRYPTION
88
64
PLAIN TEXT
FILE ENCRYPTION KEY
60
DATA DECRYPTION FIELD EXTRACTION MECHANISM
86

FIG. 4

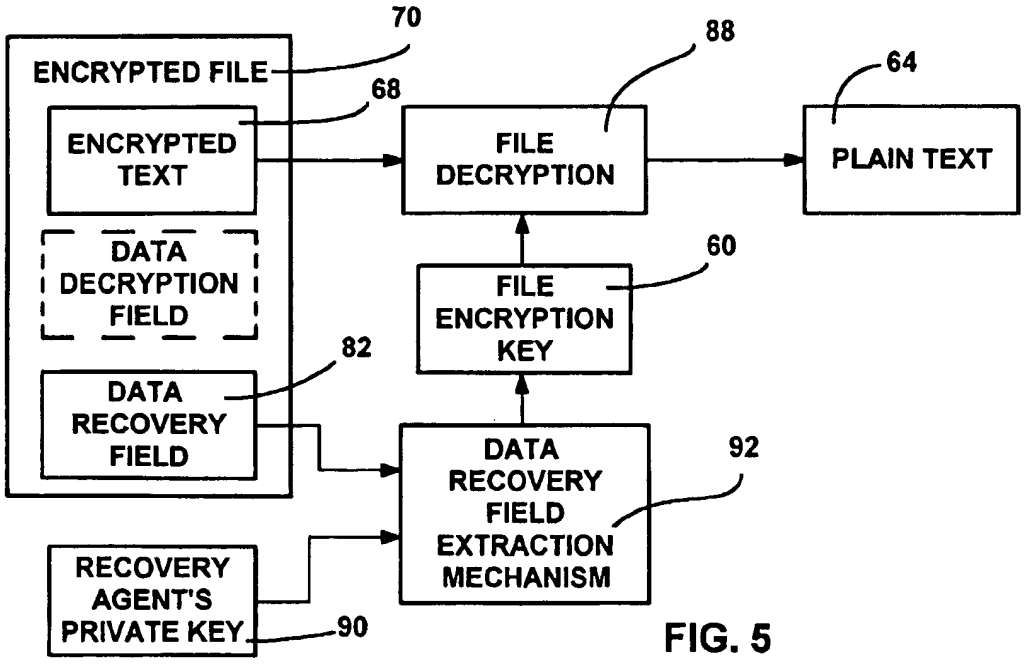

ENCRYPTED FILE
70
68
ENCRYPTED TEXT
DATA DECRYPTION FIELD
DATA RECOVERY FIELD
82
RECOVERY AGENT'S PRIVATE KEY
90
FILE DECRYPTION
88
64
PLAIN TEXT
FILE ENCRYPTION KEY
60
DATA RECOVERY FIELD EXTRACTION MECHANISM
92

FIG. 5

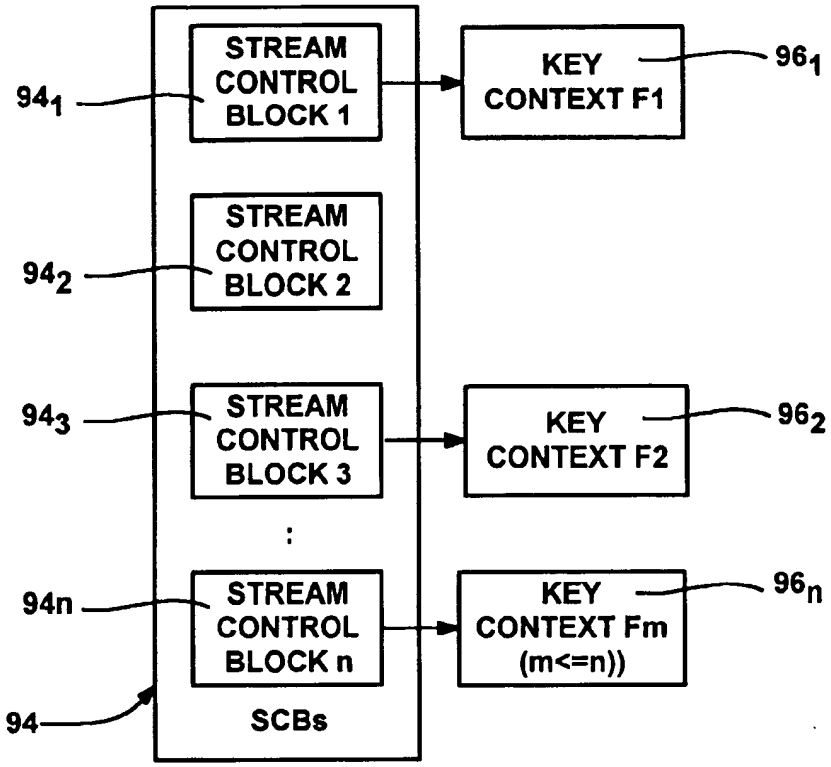

STREAM CONTROL BLOCK 1
STREAM CONTROL BLOCK 2
STREAM CONTROL BLOCK 3
STREAM CONTROL BLOCK n
SCBs
KEY CONTEXT F1
KEY CONTEXT F2
KEY CONTEXT Fm (m<=n))
94 1
94 2
94 3
94n
94
96 1
96 2
96n

FIG. 6

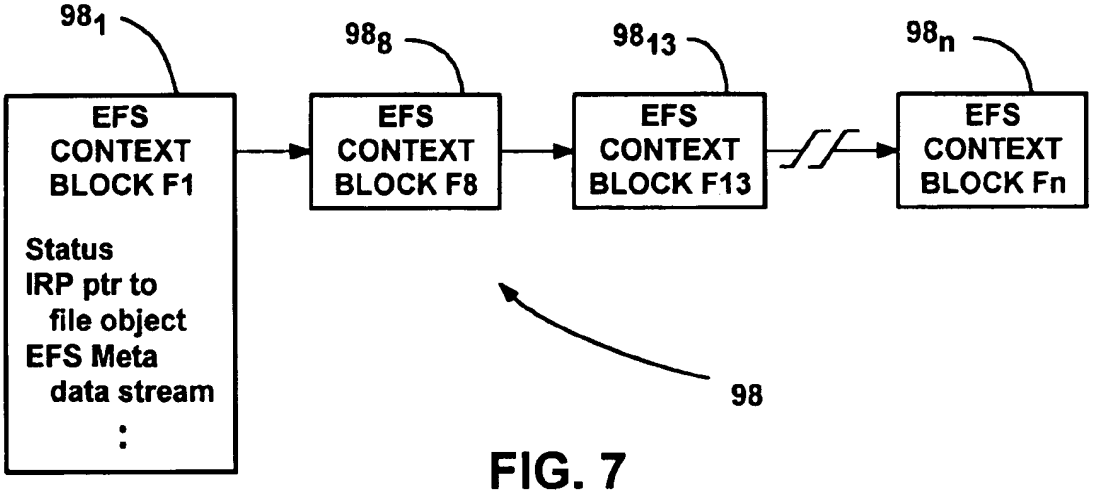

EFS CONTEXT BLOCK F1
Status
IRP ptr to file object
EFS Meta data stream
EFS CONTEXT BLOCK F8
EFS CONTEXT BLOCK F13
EFS CONTEXT BLOCK Fn
98 1
98 8
98 13
98n
98

FIG. 7

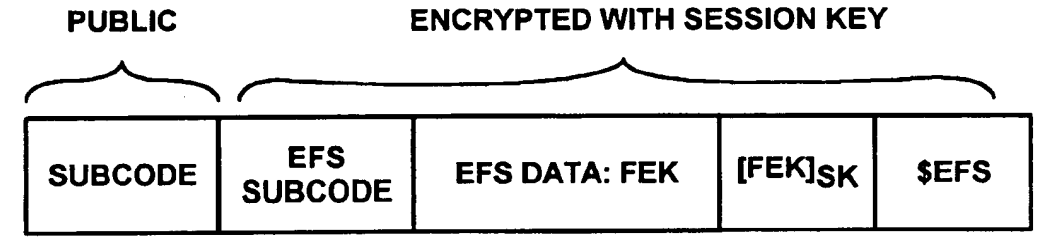

PUBLIC
ENCRYPTED WITH SESSION KEY
SUBCODE
EFS SUBCODE
EFS DATA: FEK
[FEK]SK
$EFS
100

FIG. 8

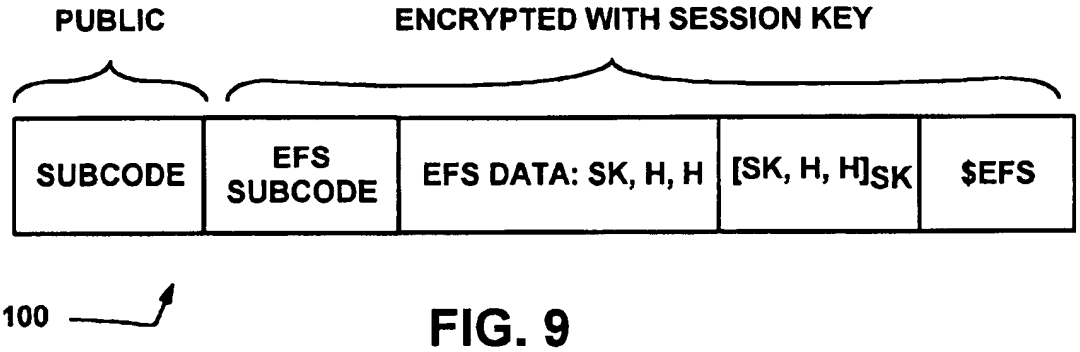

PUBLIC
ENCRYPTED WITH SESSION KEY
SUBCODE
EFS SUBCODE
EFS DATA: SK, H, H
[SK, H, H]SK
$EFS
100

FIG. 9

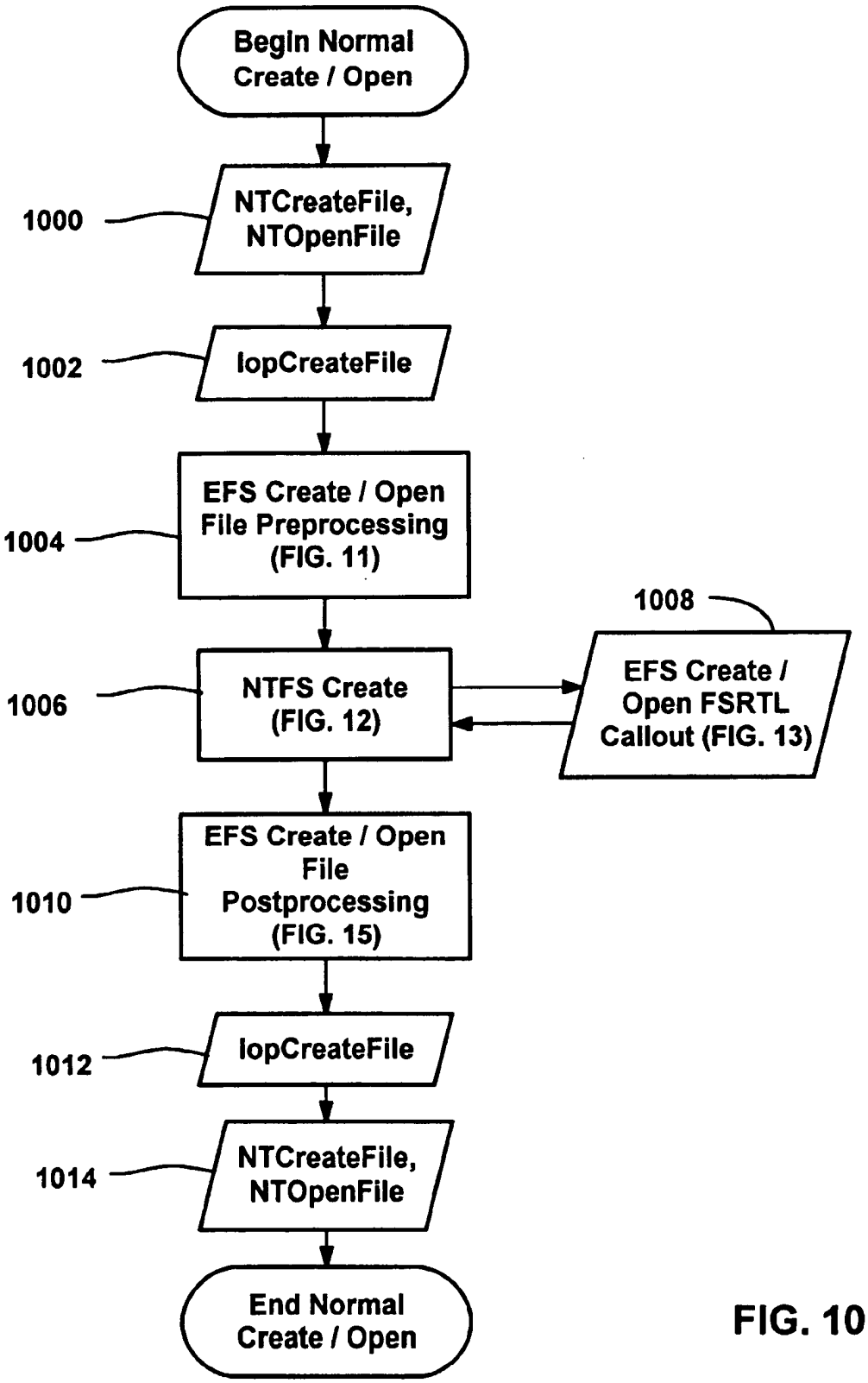

Begin Normal Create / Open
1000
NTCreateFile, NTOpenFile
1002
IopCreateFile
1004
EFS Create / Open File Preprocessing (FIG. 11)
1008
1006
NTFS Create (FIG. 12)
EFS Create / Open FSRTL Callout (FIG. 13)
1010
EFS Create / Open File Postprocessing (FIG. 15)
1012
IopCreateFile
1014
NTCreateFile, NTOpenFile
End Normal Create / Open

FIG. 10

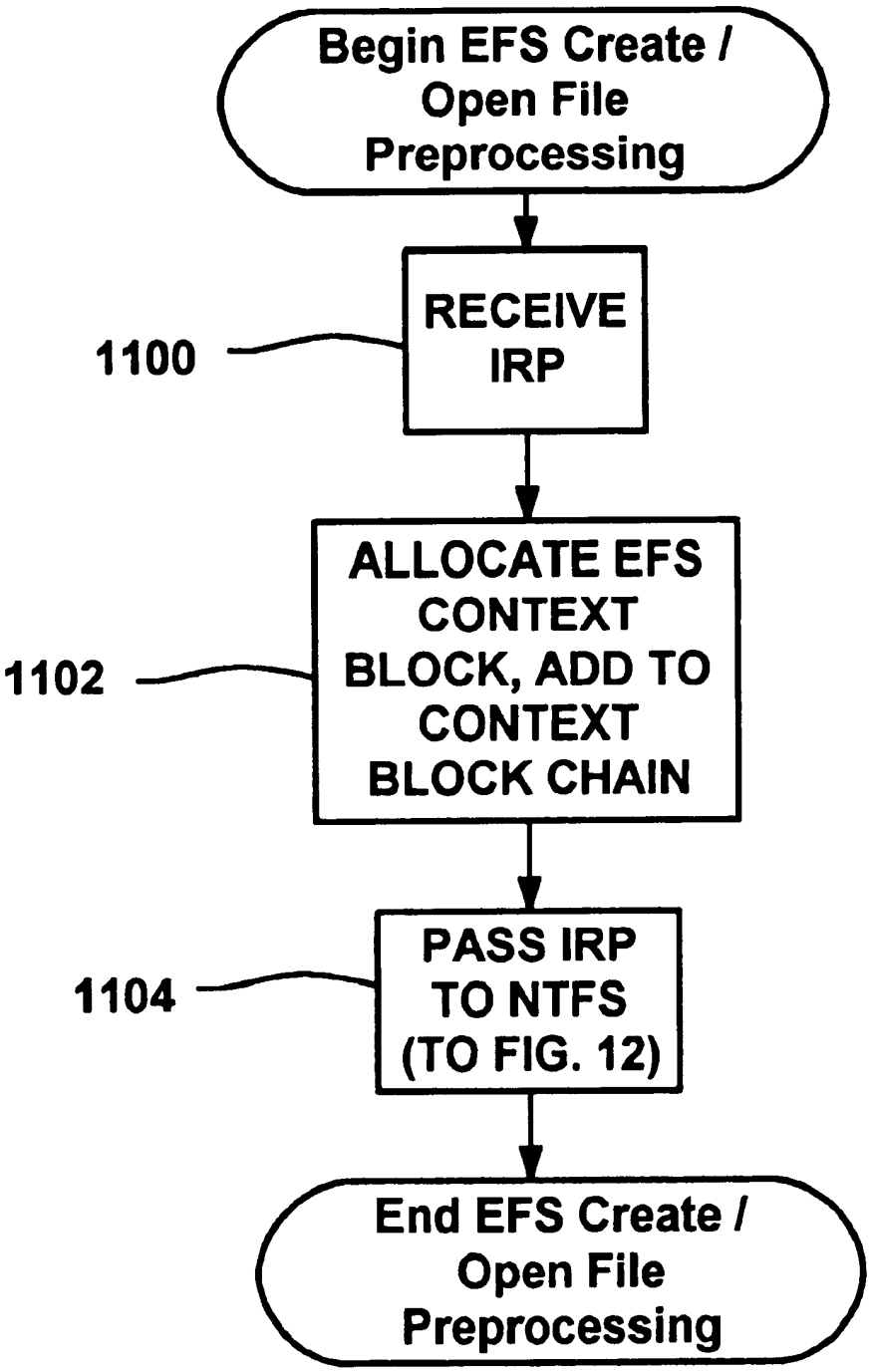

Begin EFS Create / Open File Preprocessing
RECEIVE IRP
1100
ALLOCATE EFS CONTEXT BLOCK, ADD TO CONTEXT BLOCK CHAIN
1102
PASS IRP TO NTFS (TO FIG. 12)
1104
End EFS Create / Open File Preprocessing

FIG. 11

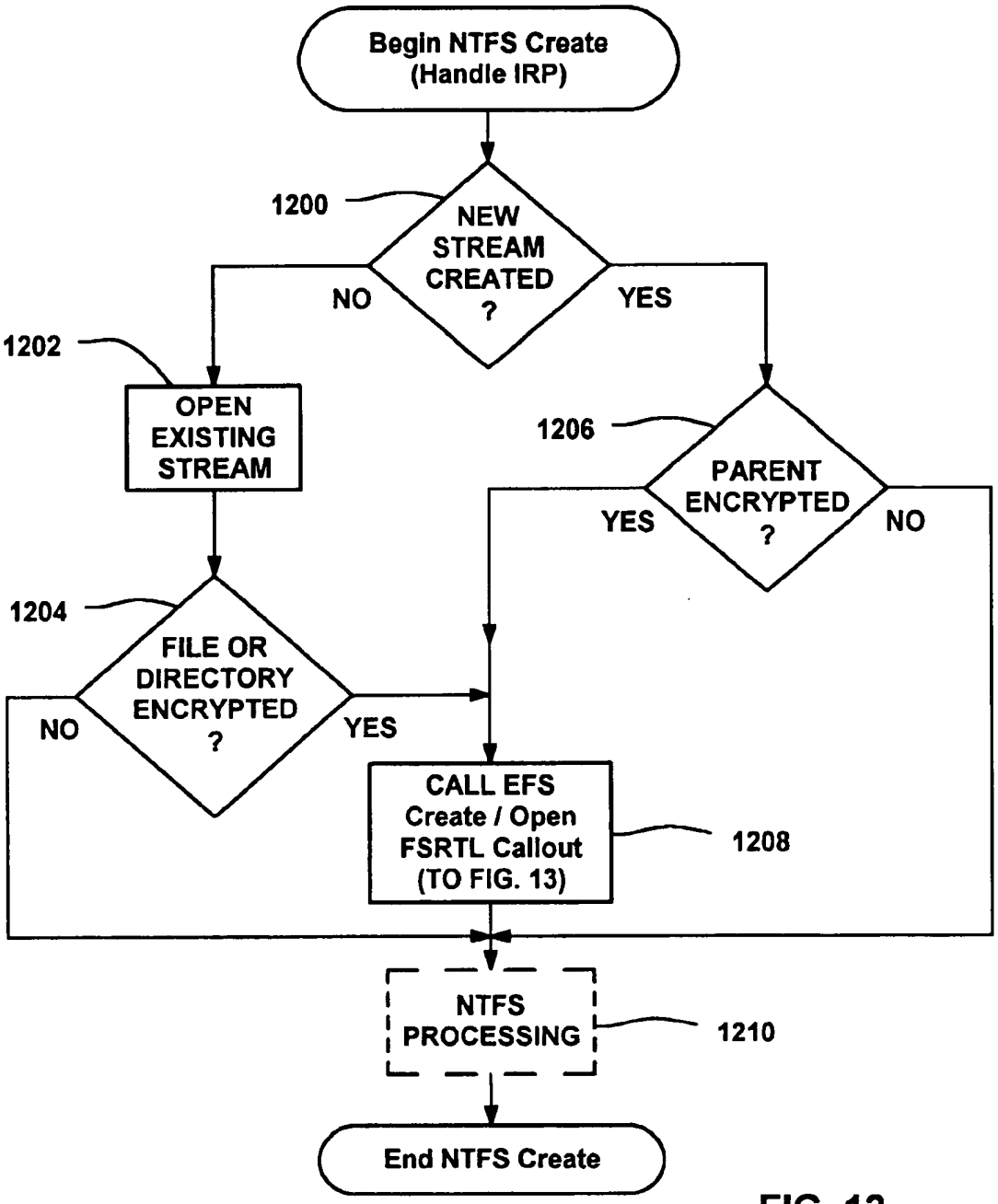

Begin NTFS Create
(Handle IRP)
1200
NEW
STREAM
CREATED
?
NO
YES
1202
OPEN
EXISTING
STREAM
1206
PARENT
ENCRYPTED
?
YES
NO
1204
FILE OR
DIRECTORY
ENCRYPTED
?
NO
YES
CALL EFS
Create / Open
FSRTL Callout
(TO FIG. 13)
1208
NTFS
PROCESSING
1210
End NTFS Create

FIG. 12

FIG. 13

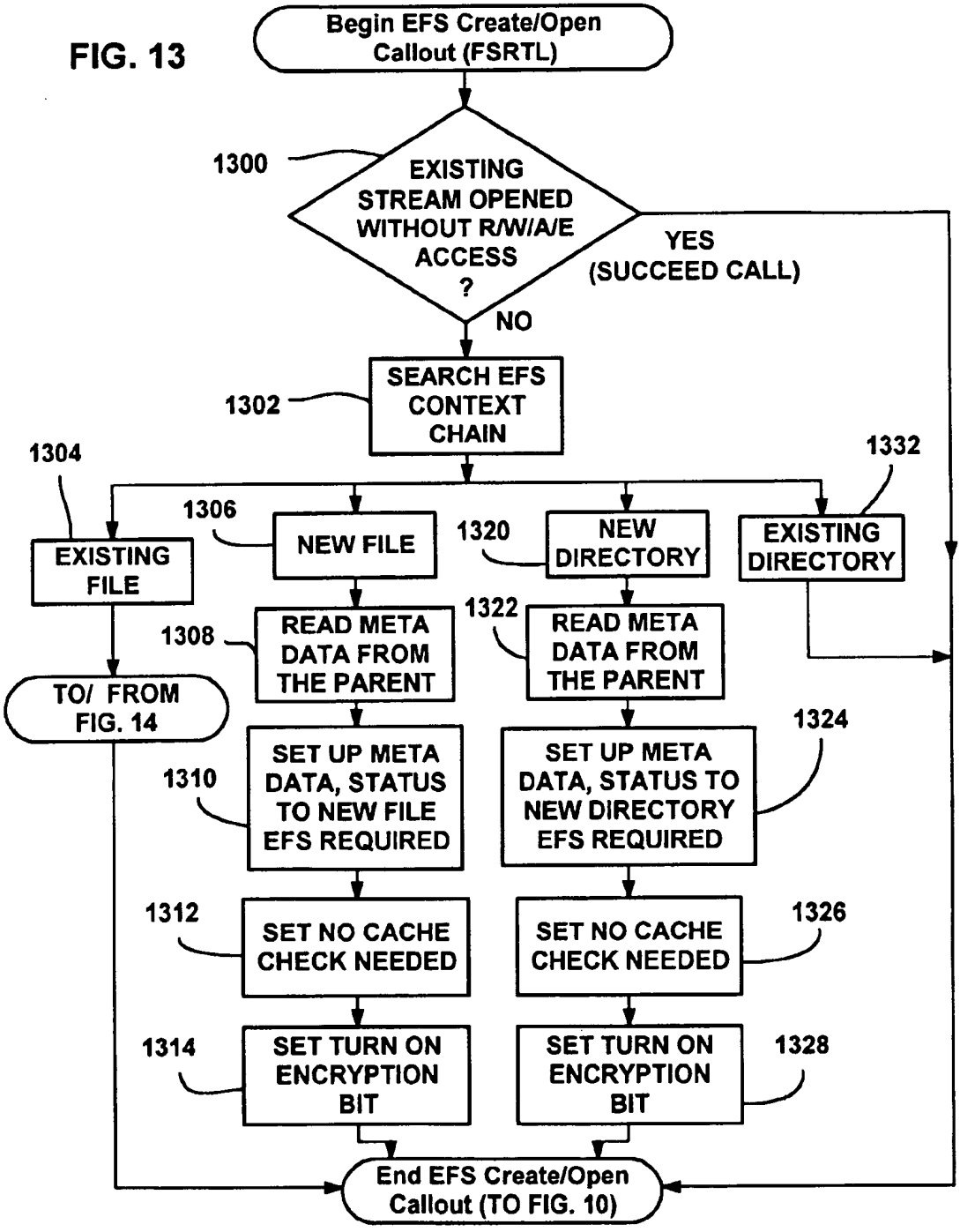

Begin EFS Create/Open Callout (FSRTL)
1300
EXISTING STREAM OPENED WITHOUT R/W/A/E ACCESS ?
YES (SUCCEED CALL)
NO
1302
SEARCH EFS CONTEXT CHAIN
1304
1306
1320
1332
EXISTING FILE
NEW FILE
NEW DIRECTORY
EXISTING DIRECTORY
1308
1322
READ META DATA FROM THE PARENT
READ META DATA FROM THE PARENT
TO/ FROM FIG. 14
1310
1324
SET UP META DATA, STATUS TO NEW FILE EFS REQUIRED
SET UP META DATA, STATUS TO NEW DIRECTORY EFS REQUIRED
1312
1326
SET NO CACHE CHECK NEEDED
SET NO CACHE CHECK NEEDED
1314
1328
SET TURN ON ENCRYPTION BIT
SET TURN ON ENCRYPTION BIT
End EFS Create/Open Callout (TO FIG. 10)

FIG. 14

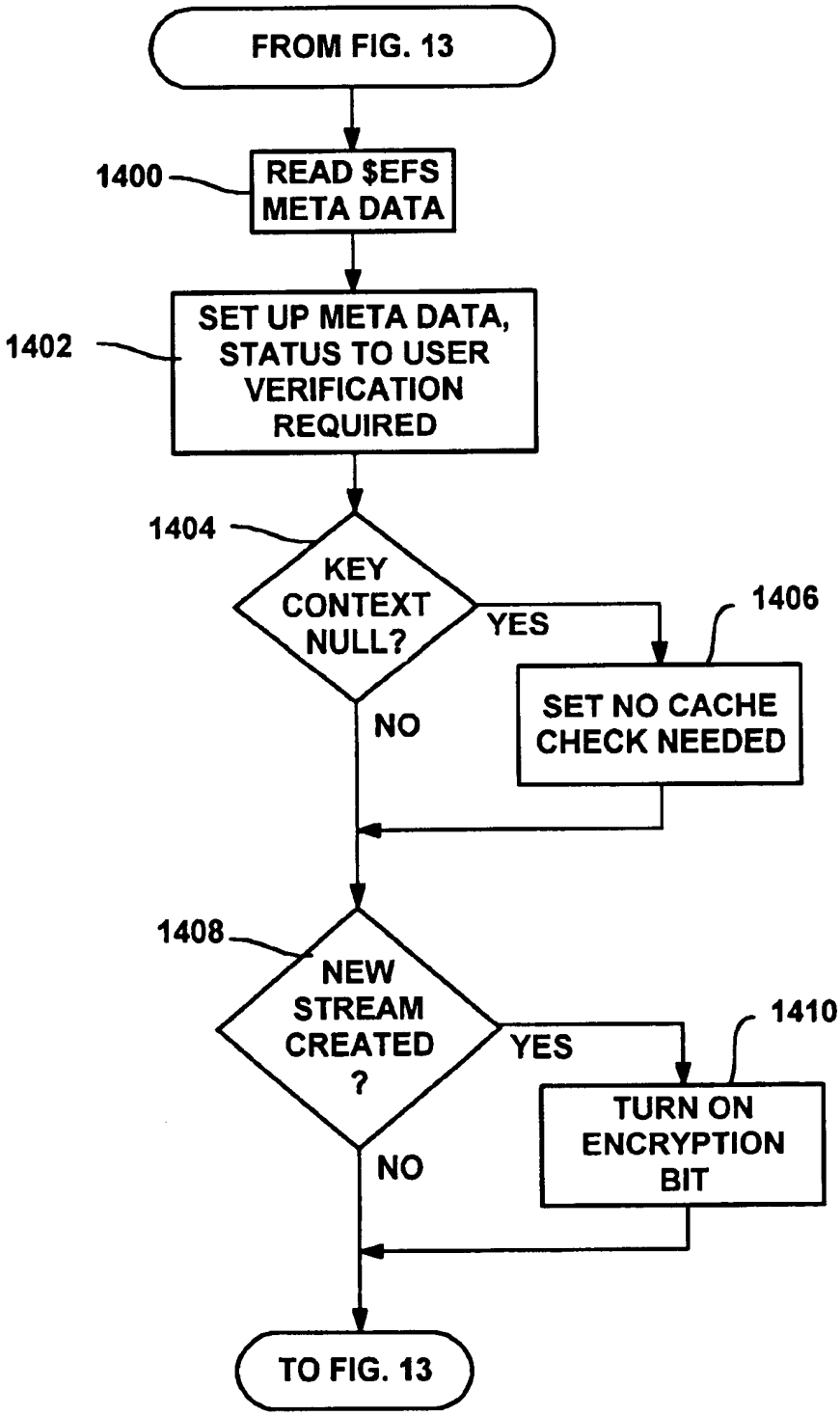

FROM FIG. 13
1400
READ $EFS META DATA
1402
SET UP META DATA, STATUS TO USER VERIFICATION REQUIRED
1404
KEY CONTEXT NULL?
YES
1406
SET NO CACHE CHECK NEEDED
NO
1408
NEW STREAM CREATED ?
YES
1410
TURN ON ENCRYPTION BIT
NO
TO FIG. 13

FIG. 15

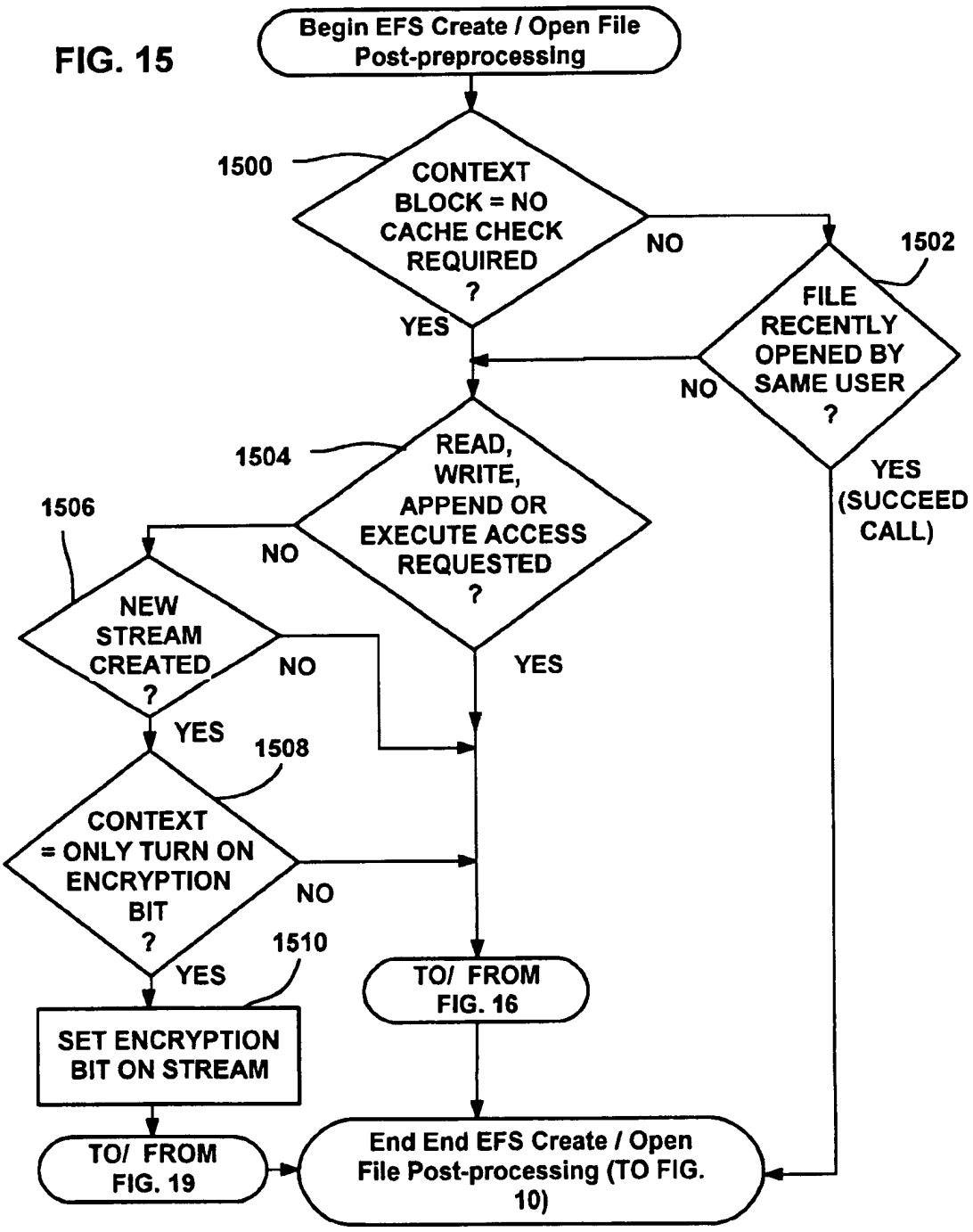

Begin EFS Create / Open File Post-preprocessing
1500
CONTEXT BLOCK = NO CACHE CHECK REQUIRED ?
NO
YES
1502
FILE RECENTLY OPENED BY SAME USER ?
NO
YES (SUCCEED CALL)
1504
READ, WRITE, APPEND OR EXECUTE ACCESS REQUESTED ?
NO
YES
1506
NEW STREAM CREATED ?
NO
YES
1508
CONTEXT = ONLY TURN ON ENCRYPTION BIT ?
NO
YES
1510
SET ENCRYPTION BIT ON STREAM
TO/ FROM FIG. 19
TO/ FROM FIG. 16
End End EFS Create / Open File Post-processing (TO FIG. 10)

FIG. 16

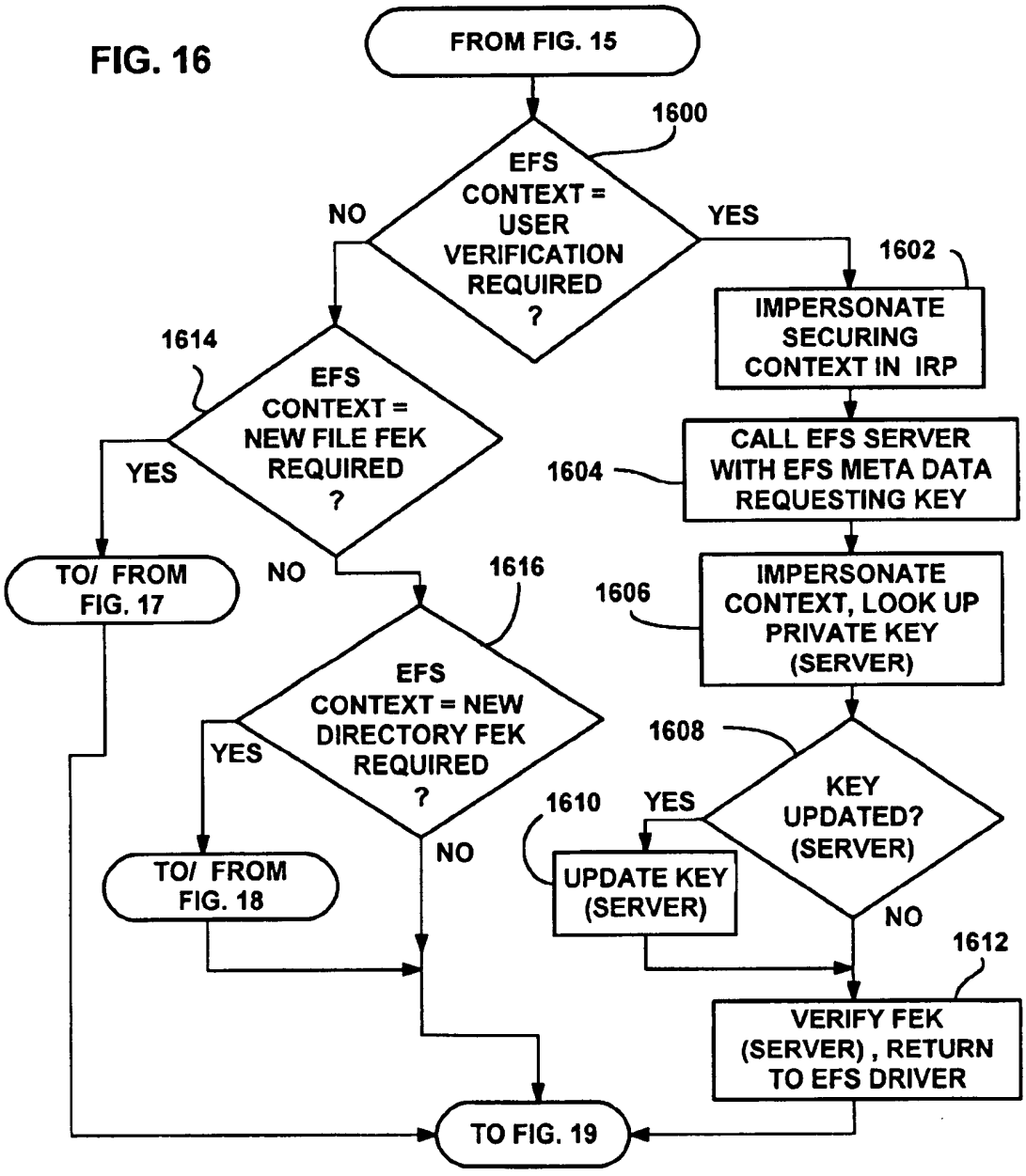

FROM FIG. 15
1600
EFS CONTEXT = USER VERIFICATION REQUIRED ?
NO
YES
1602
IMPERSONATE SECURING CONTEXT IN IRP
1604
CALL EFS SERVER WITH EFS META DATA REQUESTING KEY
1606
IMPERSONATE CONTEXT, LOOK UP PRIVATE KEY (SERVER)
1608
KEY UPDATED? (SERVER)
1610
YES
UPDATE KEY (SERVER)
NO
1612
VERIFY FEK (SERVER), RETURN TO EFS DRIVER
1614
EFS CONTEXT = NEW FILE FEK REQUIRED ?
YES
TO/ FROM FIG. 17
NO
1616
EFS CONTEXT = NEW DIRECTORY FEK REQUIRED ?
YES
TO/ FROM FIG. 18
NO
TO FIG. 19

FIG. 17

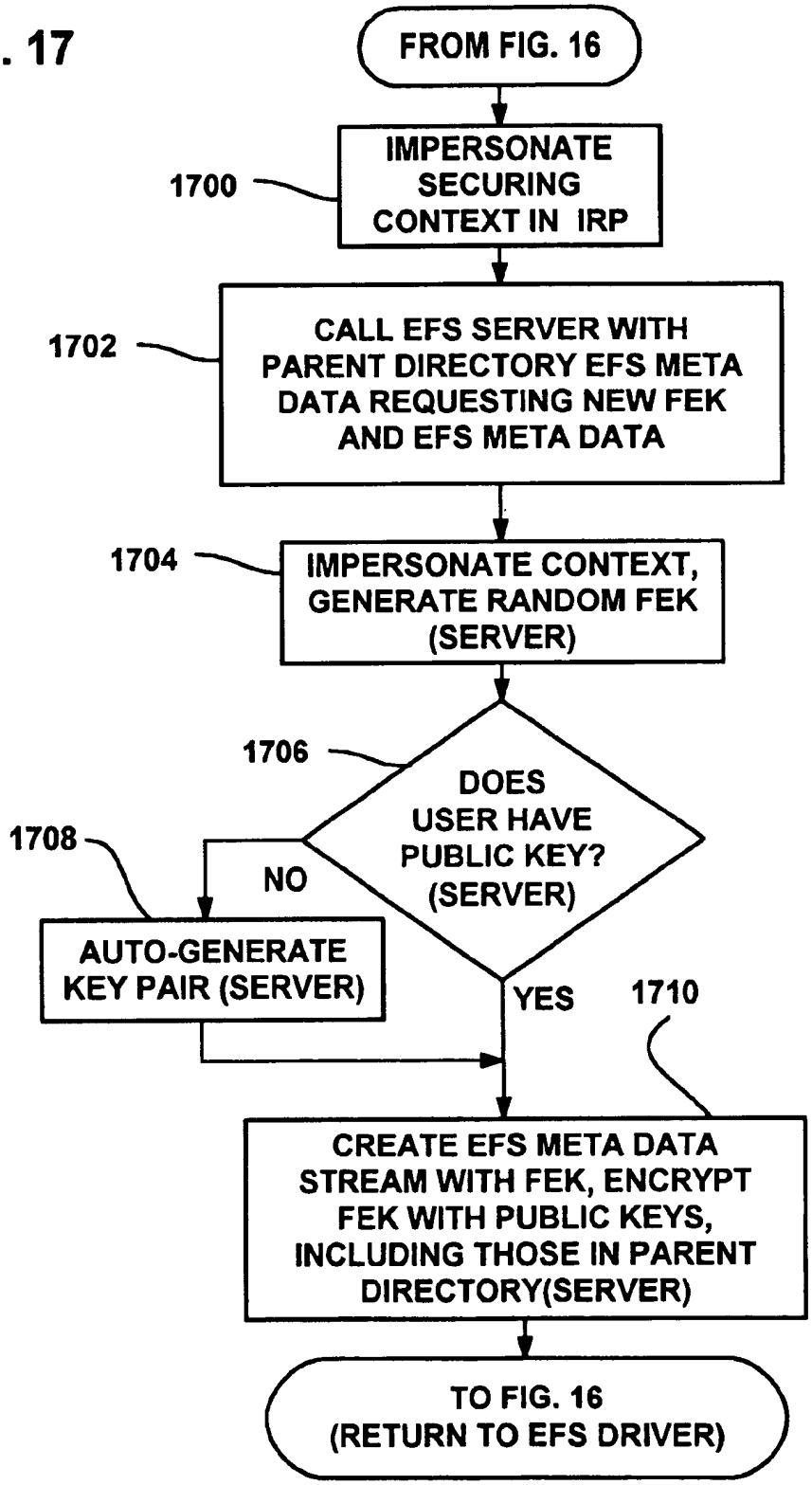

FROM FIG. 16
1700
IMPERSONATE SECURING CONTEXT IN IRP
1702
CALL EFS SERVER WITH PARENT DIRECTORY EFS META DATA REQUESTING NEW FEK AND EFS META DATA
1704
IMPERSONATE CONTEXT, GENERATE RANDOM FEK (SERVER)
1706
DOES USER HAVE PUBLIC KEY? (SERVER)
1708
NO
AUTO-GENERATE KEY PAIR (SERVER)
YES
1710
CREATE EFS META DATA STREAM WITH FEK, ENCRYPT FEK WITH PUBLIC KEYS, INCLUDING THOSE IN PARENT DIRECTORY(SERVER)
TO FIG. 16 (RETURN TO EFS DRIVER)

FIG. 18

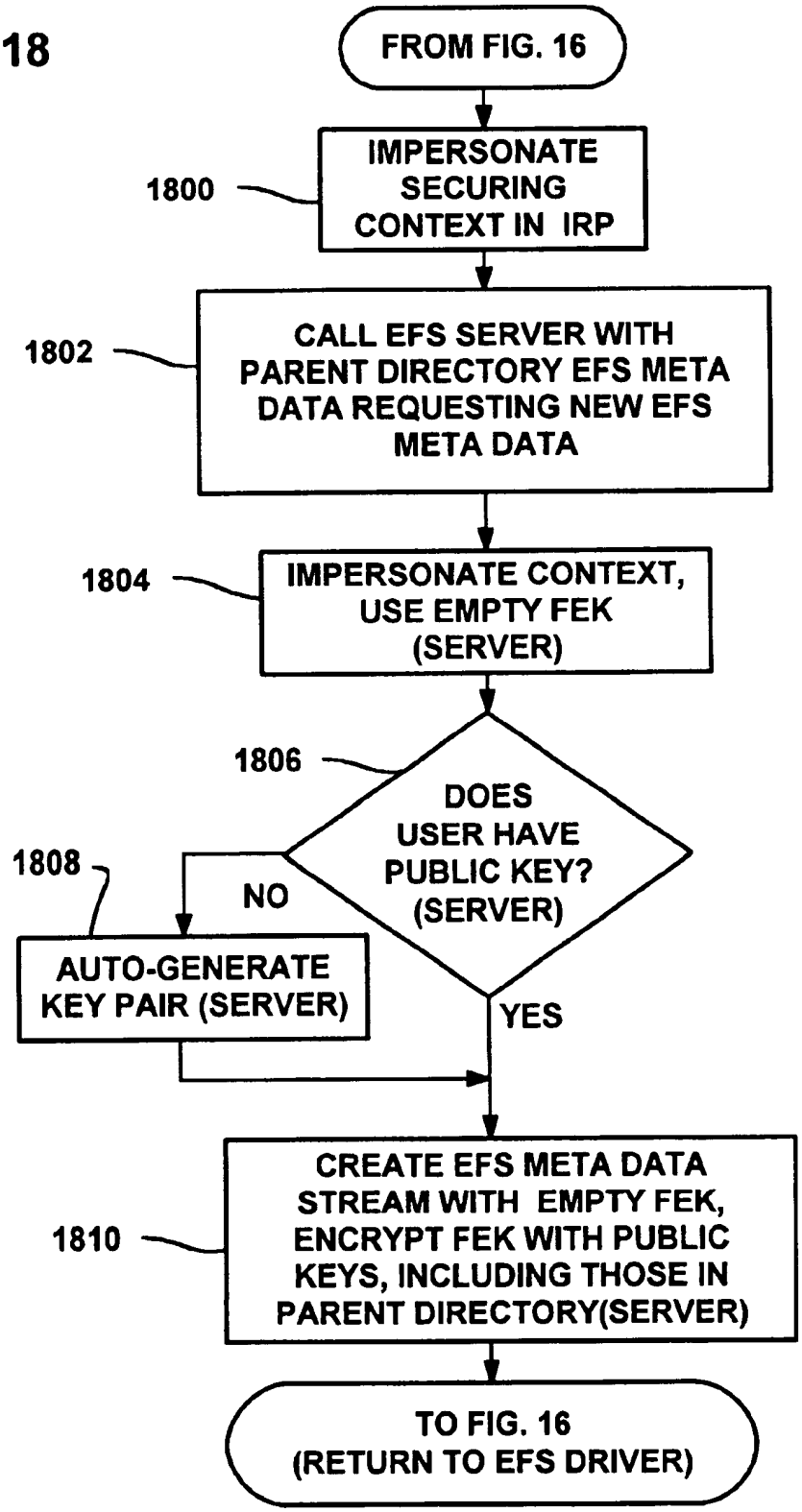

FROM FIG. 16
1800
IMPERSONATE SECURING CONTEXT IN IRP
1802
CALL EFS SERVER WITH PARENT DIRECTORY EFS META DATA REQUESTING NEW EFS META DATA
1804
IMPERSONATE CONTEXT, USE EMPTY FEK (SERVER)
1806
DOES USER HAVE PUBLIC KEY? (SERVER)
1808
NO
AUTO-GENERATE KEY PAIR (SERVER)
YES
1810
CREATE EFS META DATA STREAM WITH EMPTY FEK, ENCRYPT FEK WITH PUBLIC KEYS, INCLUDING THOSE IN PARENT DIRECTORY(SERVER)
TO FIG. 16 (RETURN TO EFS DRIVER)

FIG. 19

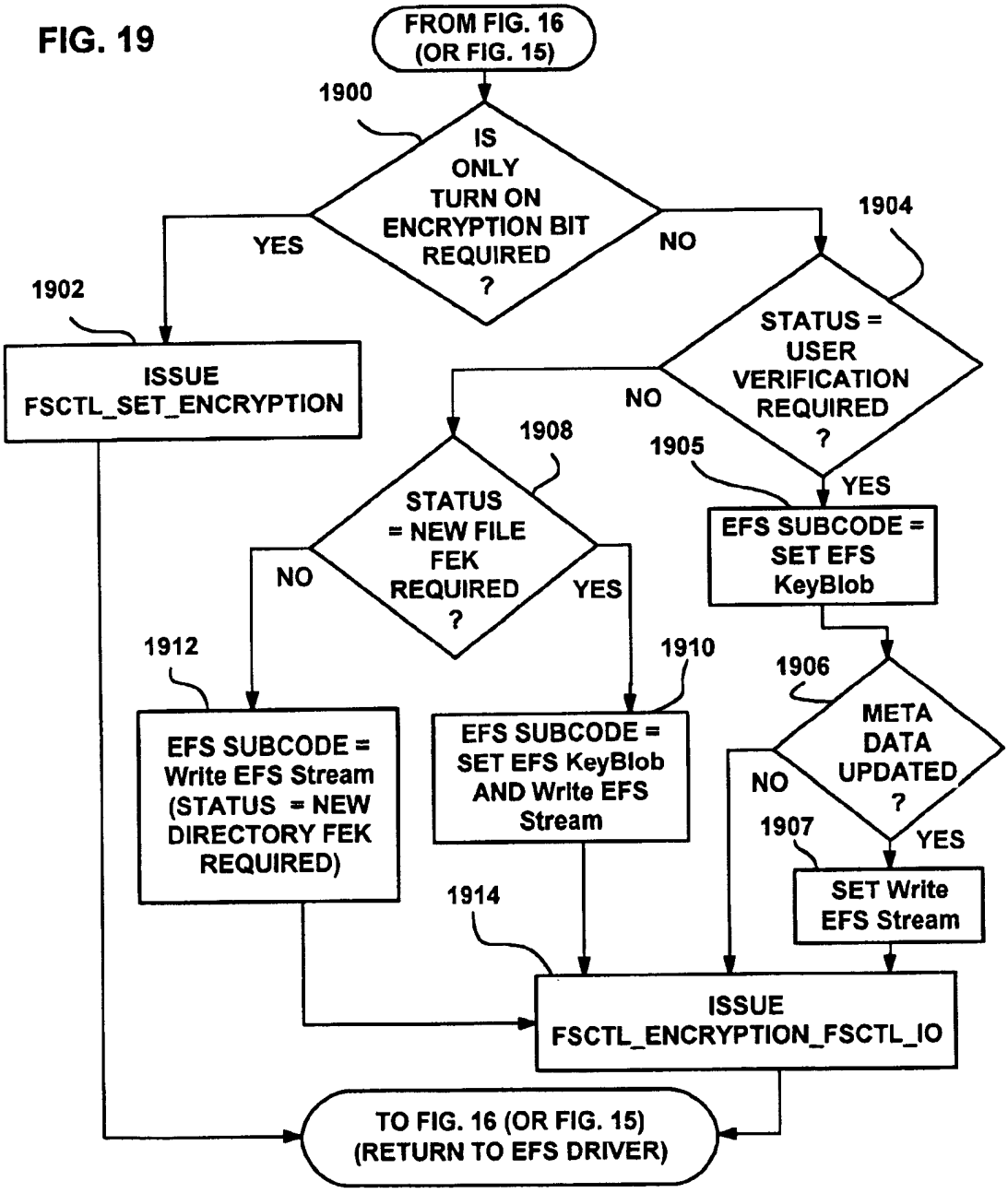

FROM FIG. 16 (OR FIG. 15)
1900
IS ONLY TURN ON ENCRYPTION BIT REQUIRED ?
YES
NO
1902
ISSUE FSCTL_SET_ENCRYPTION
1904
STATUS = USER VERIFICATION REQUIRED ?
NO
YES
1905
EFS SUBCODE = SET EFS KeyBlob
1906
META DATA UPDATED ?
NO
YES
1907
SET Write EFS Stream
1908
STATUS = NEW FILE FEK REQUIRED ?
NO
YES
1910
EFS SUBCODE = SET EFS KeyBlob AND Write EFS Stream
1912
EFS SUBCODE = Write EFS Stream (STATUS = NEW DIRECTORY FEK REQUIRED)
1914
ISSUE FSCTL_ENCRYPTION_FSCTL_IO
TO FIG. 16 (OR FIG. 15) (RETURN TO EFS DRIVER)

FIG. 20

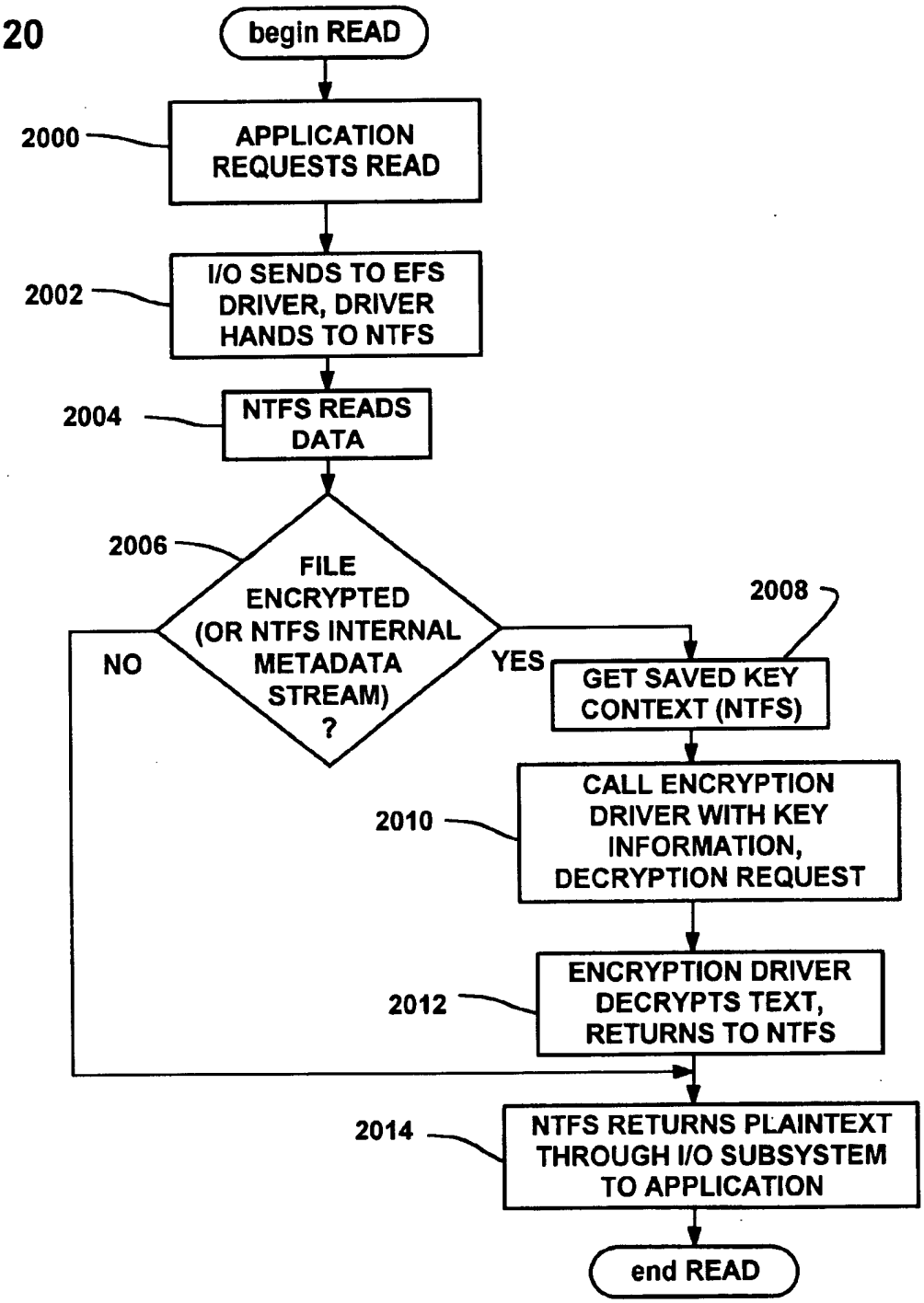

begin READ
2000
APPLICATION REQUESTS READ
2002
I/O SENDS TO EFS DRIVER, DRIVER HANDS TO NTFS
2004
NTFS READS DATA
2006
FILE ENCRYPTED (OR NTFS INTERNAL METADATA STREAM) ?
NO
YES
2008
GET SAVED KEY CONTEXT (NTFS)
2010
CALL ENCRYPTION DRIVER WITH KEY INFORMATION, DECRYPTION REQUEST
2012
ENCRYPTION DRIVER DECRYPTS TEXT, RETURNS TO NTFS
2014
NTFS RETURNS PLAINTEXT THROUGH I/O SUBSYSTEM TO APPLICATION
end READ

FIG. 21

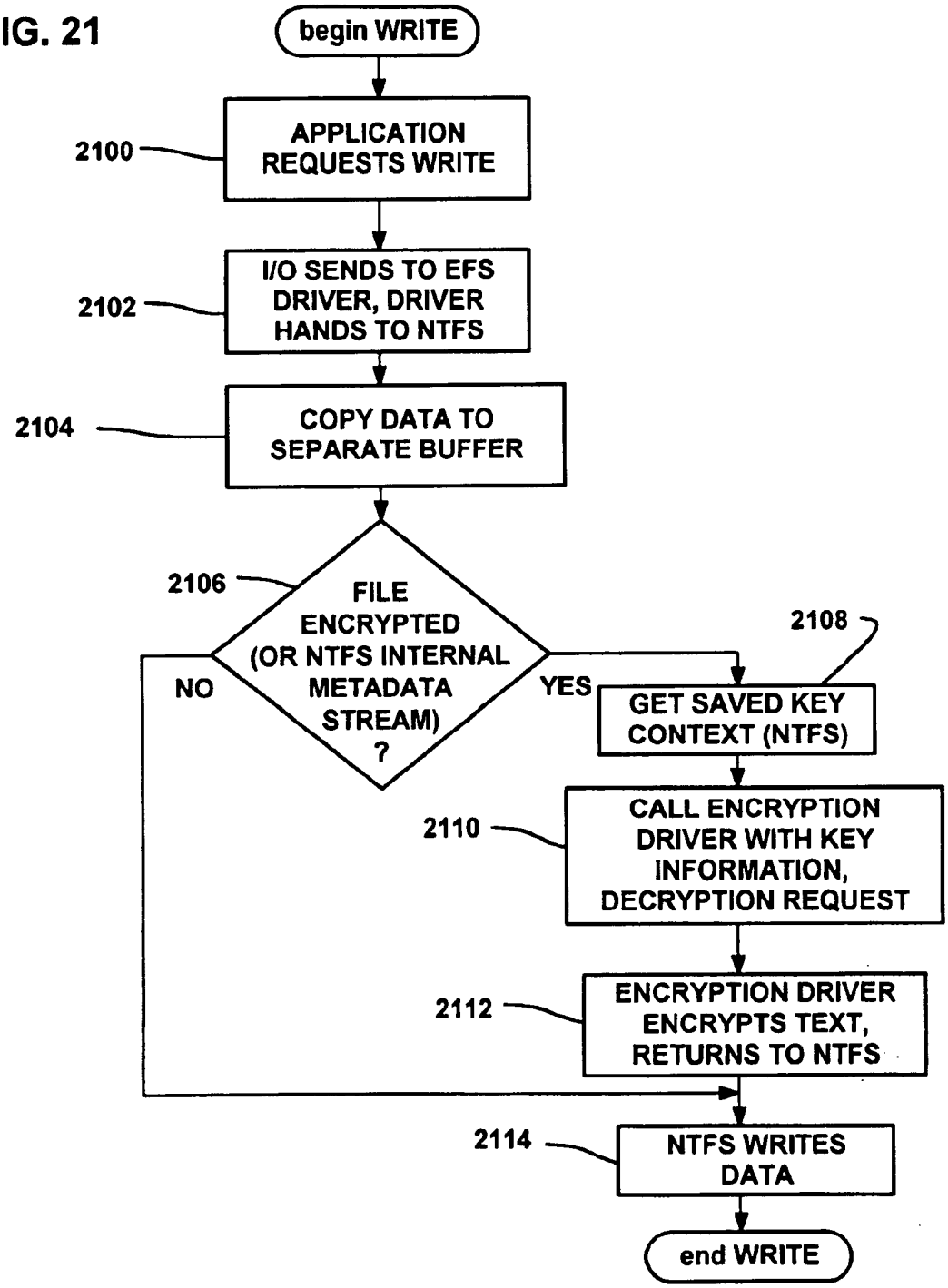

begin WRITE
2100
APPLICATION REQUESTS WRITE
2102
I/O SENDS TO EFS DRIVER, DRIVER HANDS TO NTFS
2104
COPY DATA TO SEPARATE BUFFER
2106
FILE ENCRYPTED (OR NTFS INTERNAL METADATA STREAM) ?
NO
YES
2108
GET SAVED KEY CONTEXT (NTFS)
2110
CALL ENCRYPTION DRIVER WITH KEY INFORMATION, DECRYPTION REQUEST
2112
ENCRYPTION DRIVER ENCRYPTS TEXT, RETURNS TO NTFS
2114
NTFS WRITES DATA
end WRITE

FIG. 22

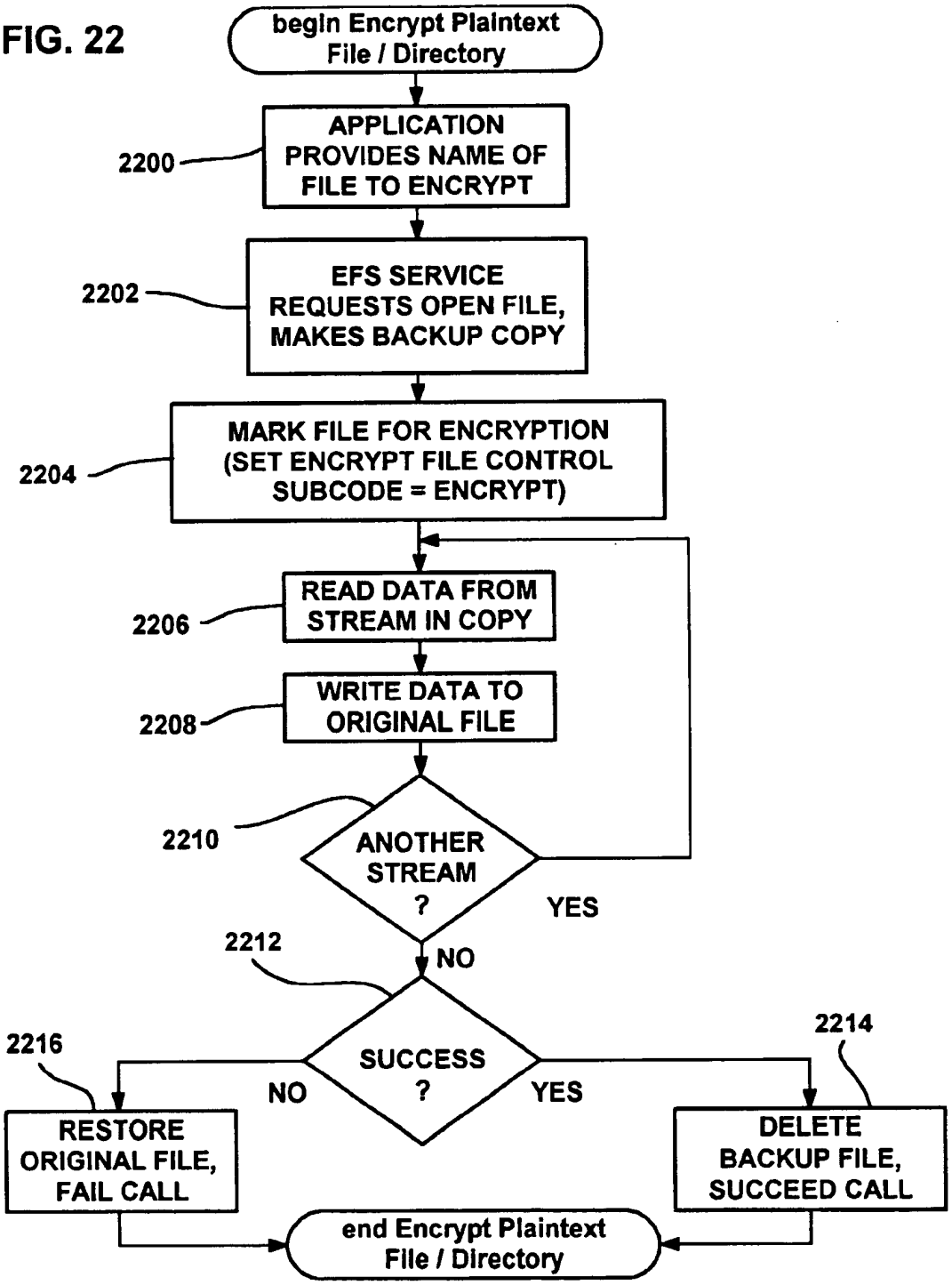
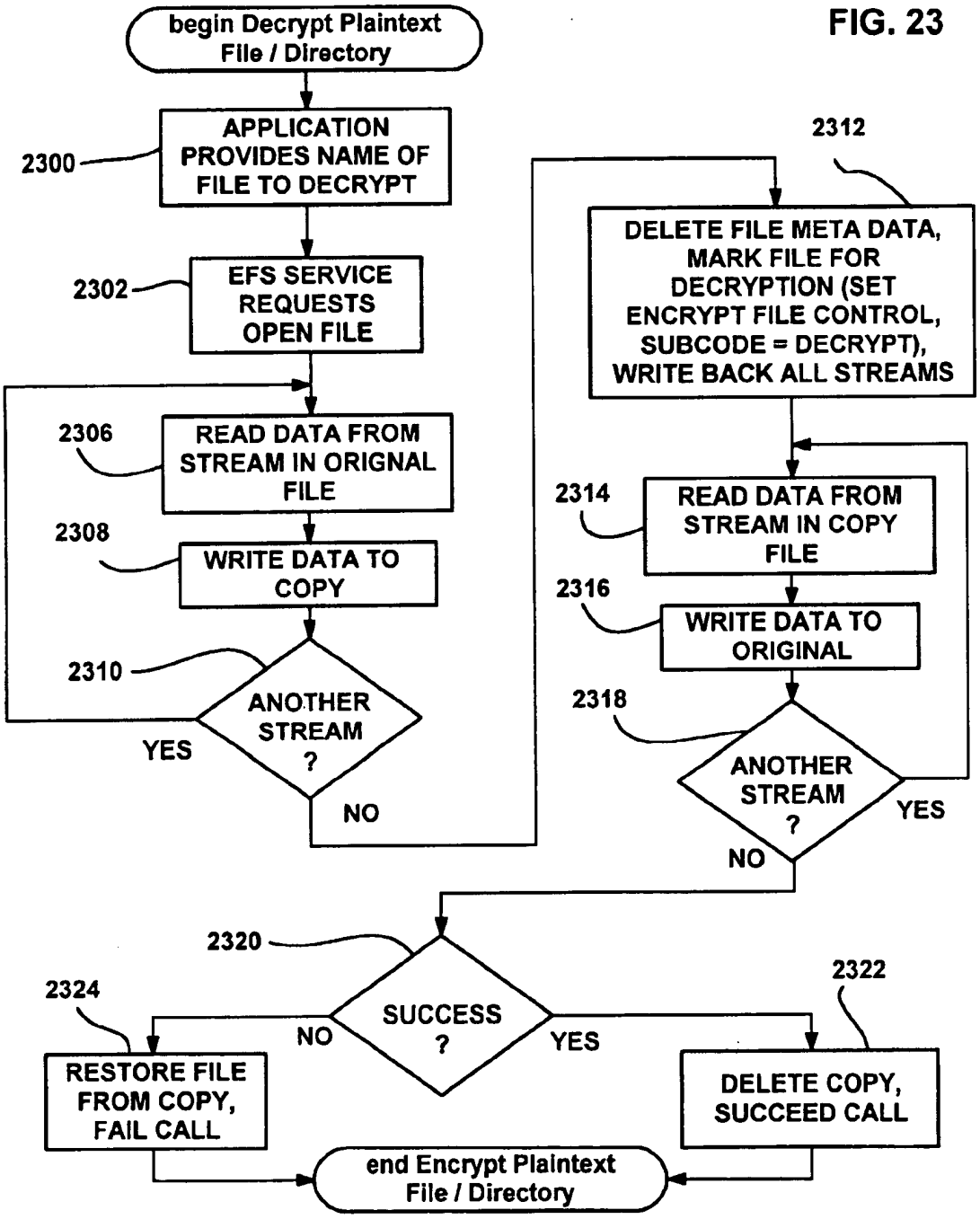

begin Encrypt Plaintext File / Directory
2200
APPLICATION PROVIDES NAME OF FILE TO ENCRYPT
2202
EFS SERVICE REQUESTS OPEN FILE, MAKES BACKUP COPY
2204
MARK FILE FOR ENCRYPTION (SET ENCRYPT FILE CONTROL SUBCODE = ENCRYPT)
2206
READ DATA FROM STREAM IN COPY
2208
WRITE DATA TO ORIGINAL FILE
2210
ANOTHER STREAM ?
YES
NO
2212
SUCCESS ?
NO
YES
2214
2216
RESTORE ORIGINAL FILE, FAIL CALL
DELETE BACKUP FILE, SUCCEED CALL
end Encrypt Plaintext File / Directory

FIG. 23


begin Decrypt Plaintext File / Directory
2300
APPLICATION PROVIDES NAME OF FILE TO DECRYPT
2302
EFS SERVICE REQUESTS OPEN FILE
2306
READ DATA FROM STREAM IN ORIGNAL FILE
2308
WRITE DATA TO COPY
2310
ANOTHER STREAM ?
YES
NO
2312
DELETE FILE META DATA, MARK FILE FOR DECRYPTION (SET ENCRYPT FILE CONTROL, SUBCODE = DECRYPT), WRITE BACK ALL STREAMS
2314
READ DATA FROM STREAM IN COPY FILE
2316
WRITE DATA TO ORIGINAL
2318
ANOTHER STREAM ?
YES
NO
2320
SUCCESS ?
NO
YES
2322
DELETE COPY, SUCCEED CALL
2324
RESTORE FILE FROM COPY, FAIL CALL
end Encrypt Plaintext File / Directory

ENCRYPTING FILE SYSTEM AND METHOD

FIELD OF THE INVENTION

The invention relates generally to computer storage techniques, and more particularly to an encrypting file system and method for computer systems.

BACKGROUND OF THE INVENTION

The protection of sensitive data is becoming a very important issue to users of computers. For example, data such as personnel records or customer credit card numbers may be stored on a computer, and if that computer (or even just the storage media) is stolen, the thief has access to the sensitive information. This is especially troublesome with laptop computers, which are frequently stolen by sophisticated thieves. Moreover, information may be misappropriated even when the storage media is not physically taken, such as when an unauthorized individual gains access to a computer (possibly when that computer is simply online) and copies information from some or all of its files. Those authorized to access the sensitive information may not even know that it has been copied.

To protect information, one type of security procedure involves encrypting the data, so that even if the data falls into the wrong hands, it cannot be read without a key. Many application level programs provide some form of such encryption. While better than nothing, however, such existing encryption schemes suffer from myriad problems.

One serious problem is that the encrypted data is still susceptible to a number of attacks. More particularly, applications providing encryption functions use password/passphrase derived keys, which are weak by definition due to dictionary attacks and the like. Moreover, attacks improve over time, particularly as hardware improves, and what was once considered safe by one application may no longer be safe.

Also, users tend to lose keys. The problem of lost keys can be eliminated by spreading the key around to multiple users, but this further compromises security. Moreover, each file may have a different password, making recall difficult. Accordingly, for convenience many users will encrypt many files with the same password key used to encrypt one file, whereby divulging a key to another person for one file often results in inadvertently giving that person the key to many other files. Moreover, in order to remove or add user access to one or more files, each file (and every copy of each file) must be decrypted and re-encrypted with the new key, and then redistributed.

Yet another significant problem is that the encryption process is inconvenient, requiring the encrypting and decrypting of each such file during each use. As a result, many typical users tend to neglect security concerns rather than bother with encrypting various files.

Another serious problem is that most applications create temporary files while a user works on a document. These temporary files are not always removed after the original is saved, leaving the sensitive data vulnerable. Also, with contemporary operating systems, application-level encryption runs in the user mode, and thus all the data, including the user's encryption key, can make into the page file, making the task of locating a key quite easy. Lastly, most file encryption applications have built-in encryption algorithms, and are therefore not extendible or flexible in supporting different or multiple algorithms that would allow users to update encryption algorithms as such algorithms improve over time, or to select from among encryption algorithms for particular data, e.g., to make a speed versus strength tradeoff based on the sensitivity of the data being encrypted.

OBJECTS AND SUMMARY OF THE INVENTION

Accordingly, it is a general objective of the present invention to provide an improved system and method for encrypting data that substantially eliminates the above-described problems.

Another objective is to provide a system and method of the above kind that is integrated into a file system.

A related objective is to provide the system and method wherein encryption and decryption work transparently to legitimate users.

In accomplishing those objectives, it is a related objective to provide a system and method as characterized above that provides for the ability to share sensitive data among more than one legitimate user, and wherein adding and removing access for a user is simple.

Yet another objective is to provide a strong cryptographic solution that addresses encrypted data recovery, such as when users lose keys.

Still another objective is to provide a system and method of the above kind that is flexible and extensible.

Briefly, the present invention provides a system and method for encrypting or decrypting data in a file. Whenever a user specifies that a file or its parent directory is encrypted, the encrypting file system receives an encryption key associated with the file. Then, when the system receives a request to write any plaintext file data to disk in an encrypted manner, the file system receives the file data, encrypts the file data into encrypted file data using the encryption key, and writes the encrypted file data to the disk. Conversely, when the system receives a request to read encrypted file data from the disk, the file system reads the encrypted file data, decrypts the read data into decrypted file data using the encryption key, and returns the decrypted file data to the user.

The encryption key is a random number encrypted by the public key of at least one user and at least one recovery agent. These keys are stored with the file, whereby the file can always be decrypted by the private key of either a user or a recovery agent.

Other objects and advantages will become apparent from the following detailed description when taken in conjunction with the drawings, in which:

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram representing a computer system into which the present invention may be incorporated;

FIG. **2** is a block diagram representing the general architecture of components of the present invention;

FIG. **3** is a block diagram conceptually representing various logical components used in the encryption of data;

FIG. **4** is a block diagram conceptually representing various logical components used in the decryption of data;

FIG. **5** is a block diagram conceptually representing various logical components used in the recovery of encrypted data;

FIG. **6** is a representation of stream control blocks associated with files, at least some of which include key context information for file encryption and decryption;

FIG. **7** is a representation of a context chain used for communicating information between encryption components for encrypted files and directories;

FIGS. 8 and 9 are representations of data structures used by certain components for communicating file information, including encryption key information, to one another;

FIG. 10 is a flow diagram showing the overall flow of control to open or create a file in accordance with one aspect of the present invention;

FIG. 11 is a flow diagram representing driver preprocessing steps generally taken as part of opening or creating a file;

FIG. 12 is a flow diagram representing steps taken by the file system to handle a file open request;

FIGS. 13–14 comprise a flow diagram representing the general steps taken by a callout to open or create an encrypted file in accordance with one aspect of the present invention;

FIGS. 15–19 comprise a flow diagram representing driver post-processing steps generally taken as part of opening or creating a file;

FIG. 20 is a flow diagram representing steps taken by the various components to handle a file read request;

FIG. 21 is a flow diagram representing steps taken by the various components to handle a file write request;

FIG. 22 is a flow diagram representing steps taken by the various components to handle a user request to encrypt a stored plaintext file; and

FIG. 23 is a flow diagram representing steps taken by the various components to handle a user request to decrypt a stored encrypted file.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

GENERAL ARCHITECTURE

Turning to the drawings and referring first to FIG. 1, there is shown a computer system 20 generally designated 20 into which the present invention may be incorporated. The illustrated computer system 20 may be a server, a workstation, or a combination thereof, and may be connected in a known manner to one or more other computer-based resources. Of course, as will become apparent, the invention is not limited to any particular type of computer or network architecture, and may be incorporated into a stand-alone personal computer or the like.

As shown in FIG. 1, the computer system 20 includes a processor 22 connected to a memory 24 having an operating system 26 loaded therein. Preferably, the operating system 26 is Microsoft Corporation's Windows NT operating system. The computer 20 has a file system 28 such as the Windows NT File system 28 (NTFS 28) associated with or included within the operating system 26. However, as can be appreciated, the present invention is not limited to any particular operating system and/or file system, but for clarity the present invention will be hereinafter described with reference to Windows NT and NTFS 28. At least one application program 30 in the memory 24 interfaces with the operating system 26 and the file system 28 through application programming interfaces (APIs) 32.

The computer system 20 also includes input-output (I/O) circuitry 34 for connecting the computer system 20 to one or more networked devices, to one or more input devices 36 such as a keyboard and/or mouse, and/or to one or more output devices 38 such as a monitor and/or speakers. The computer system 20 also includes a non-volatile storage device 40 such as a hard disk drive. As can be appreciated, the non-volatile storage device 40 may also operate in conjunction with the random access memory of the memory 24 to provide a large amount of virtual memory via swapping techniques.

The file system 28 connects through a device driver 42 to communicate with the non-volatile storage device 40, to manage the files thereon, and generally contains methods for (1) storing, referencing, sharing and securing files, (2) accessing file data and (3) maintaining file integrity. Notwithstanding, there is not always a clear distinction between a file system 28 and its associated operating system, particularly with those file systems 28 contained within an operating system. Accordingly, it is understood that any or all of the processes or steps attributed herein to the file system 28 may alternatively be performed by the operating system 26, and vice-versa.

The non-volatile storage 40 stores a number of files 44_1–44_n, which, when NTFS serves as the file system 28, have their data organized in attribute data streams. An NTFS file control block (FCB) associated with each file maintains information identifying the data streams belonging thereto. Windows NT and NTFS 28 are described in the texts, Inside Windows NT, by Helen Custer, Microsoft Press (1993) and Inside the Windows NT File System, Helen Custer, Microsoft Press (1994), which are herein incorporated by reference in their entireties. As shown in FIG. 1 and as described below, in accordance with the present invention, at least some of the files 44_1–44_n are stored with encrypted data.

EFS DRIVER

In accordance with one aspect of the present invention, to encrypt and decrypt the files, as best shown in FIG. 2, an encrypting file system is provided, comprising an Encrypting file system (EFS) driver 46, an EFS runtime library (FSRTL 48) 48 and an EFS service 50. The EFS driver 46 is layered on top of NTFS 28, and communicates with the EFS service 50 to request file encryption keys, decryption fields (described below) and other key management services. Via NTFS 28, the EFS driver 46 passes this information to the FSRTL 48 to perform various file system operations such as open, read, write and append, as described below. In keeping with the invention, once a file is encrypted or stored in an encrypted directory, the encryption and decryption processes operate transparently to the user.

The EFS driver 46 is loaded dynamically by NTFS 28 when NTFS 28 is initialized. NTFS 28 causes the EFS driver 46 to attach to NTFS 28 (i.e., be inserted into the layered driver chain) for a particular volume when NTFS 28 first receives an I/O request packet to create/open a file on that volume, or when NTFS 28 receives an I/O request packet with a file control operation attempting an encryption related operation on a file on that volume. During initialization of the EFS driver 46, EFS registers file system runtime library callback routines with the NTFS 28. As described below, NTFS 28 uses these FSRTL 48 routines to call back to obtain file encryption related services.

The EFS driver 46 provides the support to communicate with the user mode EFS service 50 running as part of the security subsystem. During initialization, the EFS driver 46 communicates with the EFS service 50 using a GenerateSessionKey interface, to establish a symmetric session key that is used to communicate securely between the driver 46 and the EFS service 50. All data communicated between the two is encrypted using this session key. This session key is also used by callouts to the FSRTL 48 to decrypt I/O controls from the EFS service 50.

During every open of an encrypted file, the EFS driver 46 communicates with the EFS service 50 by passing it the file metadata, including the data decryption and data recovery fields, (FIGS. 3–5, described below), to get back the file

encryption key and any updates to the file metadata. The file metadata may be updated because the user may change to a new key, or the recovery agent's keys might get updated. The EFS driver **46** passes this information to FSRTL **48**.

During encryption of a plaintext file/directory or creation of a new encrypted file, the EFS driver **46** communicates with the EFS service **50** to get a new file encryption key, and encryption metadata for the encrypted file. The EFS driver **46** also passes this information to the FSRTL **48**.

EFS FSRTL

The FSRTL **48** is a module that implements NTFS callouts to handle various file system **28** operations such as reads, writes, and opens, on encrypted files and directories, as well as operations to encrypt, decrypt, and recover file data when it is written to or read from disk. To this end, the present invention provides a callout mechanism including an interface between NTFS **28** and the FSRTL **48**. As described in more detail below, this interface is generic to any appropriate library (and driver) that transform data, including the ones described herein that encrypt data, and thus the interface between NTFS **28** and FSRTL **48** is more accurately referred to as a data transformation interface **52**. For example, an indexing driver could use this interface to monitor all writes to disk and develop an index based on those writes. However, as can be appreciated, a dedicated encryption interface may be alternatively provided.

Operations between the EFS driver **46** and FSRTL **48** include writing EFS attribute data (decryption data and recovery fields) as file attributes, and communicating a file encryption key computed in the EFS service **50** to FSRTL **48**, such that it can be set up in the context of an open file. This file context is then used for transparent encryption and decryption on writes and reads of file data to and from the non-volatile storage **24**.

The data transformation interface **52** is capable of interfacing to any engine or driver that transforms the data in virtually any way, but for purposes herein the interface **52** will be described as interfacing the EFS driver **46** to the file system **28** for accomplishing data encryption. Notwithstanding, the data transformation interface is not limited to data encryption, but is appropriate for accomplishing virtually any type of data alteration. At present, however, this transformation model supports in-place data transformation wherein the data takes at least no more space than the original plain text. In any event, the EFS driver **46** registers these callbacks with the file system **28**, whereby the file system **28** uses the registered EFS callback functions at appropriate times to carry out the various encrypting and decrypting tasks that the user requests.

Although not necessary to the invention, for convenience, the FSRTL **48** is stored in a common file with the EFS driver **46**. Indeed, although the EFS driver **46** and FSRTL **48** are implemented as a single component, they do not communicate directly, but instead use the NTFS file control callout mechanism, i.e., the EFS driver **46** can effectively call the FSRTL **48**. The use of the NTFS callout mechanism ensures that NTFS **28** participates in all file operations, which avoids conditions such as where two users are locked, each user waiting for the release of the other's file.

The data transformation interface **52** includes a number of function pointers, or callbacks. A first callback which the file system **28** uses, the FileCreate callback, tells the registered EFS functions that a stream is being created or opened. The actions that EFS driver **46** takes at this point (e.g., determining if a user has access if the file is an existing file or getting the metadata stream for a new file) are described in more detail below.

When an application opens or creates a file, the I/O subsystem **56** determines the file is of a certain file system, e.g., an NTFS **28** file, and passes the request on to NTFS **28**. NTFS **28** determines whether EFS may be interested in the file, e.g., if the file is created in an encrypted directory or if a stream is created or attached to an encyrpted file. IF NTFS **28** determines that the file is of interest to EFS, and sees that the EFS driver **46** has registered itself as a driver therewith, NTFS **28** calls a registered EFS function, i.e., the FileCreate callback. If the request is a file open request on an existing file, FSRTL **48** reads the file metadata from the file attribute and fills up a context block (e.g., block 98_1 of FIG. **7**, previously allocated by the EFS driver **46**, as described below) to pass back that information to the EFS driver **46**. When the call returns from NTFS **28**, the EFS driver **46** takes the metadata information and communicates with the EFS service **50** to extract the file encryption key **60** from the metadata. This information is then returned by the EFS driver **46** to NTFS **28** by another FSRTL **48** interface, FileControl, described below, which sets up a key context **96** on the file being opened. This key context **96** is thereafter retained by NTFS **28** for future calls to the EFS driver **46** until the file is closed. If the file metadata is updated, the updated metadata is also re-written to the attributes by the registered EFS functions through NTFS callbacks.

If a new file is created, the FileCreate call results in the FSRTL **48** filling up the context buffer 98_1 with a request for a new file encryption key and metadata. The FSRTL **48** then passes the context buffer 98_1 back to the EFS driver **46**. The EFS driver **46** takes this information and communicates with the EFS service **50** to obtain a new file encryption key and new file metadata from the EFS service **50**. Using a file control callback (described below), the EFS driver **46** returns this information to the FSRTL **48**, whereby, using NtOfs function calls, the FSRTL **48** sets up the key context **98** on the file being created and writes the file metadata. The NtOfs API is a set of NTFS **28** function calls that allow the EFS driver **46** to call into the file system **28** to manipulate the data streams containing the encryption meta data.

Another callback, FileSystemControl_1, is called by NTFS **28** in response to the EFS driver **46** request when a user is setting the encryption state of a file (EFS_SET_ENCRYPT), either marking it as encrypted or decrypted. In response, NTFS **28** sets or clears the encryption bit, and the EFS driver **46** generates any necessary key storage. EFS_SET_ENCRYPT also originates in the EFS service **50** when a plaintext file begins to be encrypted, whereby the file state is modified such that no other operations are allowed on the file until the encryption is completed.

NTFS **28** also calls the FileSystemControl_2 interface with various encryption driver-specific file control requests from the EFS driver **46**. Note that NTFS **28** takes no action with these callbacks other than to simply pass the call to the FSRTL **48**. The file control requests include EFS_SET_ATTRIBUTE, which comes from the EFS filter driver **46** when it wants to write new or updated file metadata, and EFS_GET_ATTRIBUTE, which may come from the EFS driver **46** or a user mode application **30** to query the file metadata. The information includes the list of user public keys and recovery agent public keys (described below) that are used to encrypt the file encryption key. Another request, EFS_DECRYPT_BEGIN, comes from the EFS service **50** when it starts decrypting an encrypted file. In response, the state of the file is modified such that no other operations are allowed on the file until the decryption is completed. EFS_DEL_ATTRIBUTE is a request originating in the EFS service **50** when it finishes decrypting an entire encrypted

file, and wants to delete the file metadata and associated attribute. The EFS_ENCRYPT_DONE request also comes from the EFS service **50** when it successfully completes the file encryption. The file state is modified to allow any operations from this point on. EFS_OVERWRITE_ATTRIBUTE comes from the EFS service **50** when an encryption file is restored from its backup format. The EFS service **50** supplies the file metadata that needs to overwrite any existing metadata on the file. This request is also associated with the deletion of any key context **96** associated with that file, such that no reads or writes can proceed while the file is being restored.

The FileSystemControl_2 interface is also called by the file system **28** in response to the FSCTL_ENCRYPTION_FSCTL_IO, also described below. This provides a means for the EFS driver **46** to have NTFS **28** call the EFS driver **46** (itself), such as when NTFS **28** recognizes that a file is in a certain state corresponding to a state for which the EFS driver **46** is waiting.

The file system **28** directly uses the callback, AfterReadProcess after it has read some data from the disk for an encrypted file, and before returning it to the user. The AfterReadProcess function decrypts the data on the fly in response to this callback. The read operation is described in more detail below with respect to FIG. **20**.

Conversely, BeforeWriteProcess is called by the file system **28** before it writes some data to the disk for an encrypted file. The function encrypts the data as a result of this callback. The write operation is described in more detail below with respect to FIG. **21**.

The CleanUp callback is called by the file system **28** when NTFS **28** is freeing any resources associated with a stream. At this time, the EFS driver **46** frees up any memory resources it was using, such as to store keys and the like. When NTFS **28** receives its last close on a stream, NTFS **28** performs its normal operations to free up anything stored in memory to keep track of this open file, including the key context **96**. In addition, the file system **28** calls the EFS driver **46** with the context block **98**, giving it the opportunity to free up any memory it was consuming for this file, e.g., the context block **98**.

The AttachVolume callback is called by a file system **28** (during the first user operation involving encryption), as described above. In response, the EFS driver **46** notifies the I/O subsystem that it wants to attach to the device object representing that volume, thereby logically placing itself above NTFS **28** for that volume so that the I/O subsystem will pass information to the EFS driver **46** first. DismountVolume is called by a file system **28** when a volume is being dismounted, either because a user wishes to eject or remove the drive, or because the system is being shut down. In response to the DismountVolume call, an encryption driver may free any memory resources that were allocated during the AttachVolume callback. However, it should be noted that the EFS driver **46** ordinarily detaches itself and frees any resources when notified by the I/O subsystem of a volume dismount, but the DismountVolume callback is provided anyway to provide additional flexibility.

EFS SERVICE **50**

The EFS service **50** is part of the Windows NT security subsystem. As represented in FIG. **2** as EFS service **50**/Driver Communication **54**, the EFS service **50** uses the existing local procedure call communication port between a Local Security Authority (LSA) and the kernel mode security reference monitor to communicate with the EFS driver **46**. In the user mode, the EFS service **50** **24** interfaces with Microsoft's Cryptography API, CryptoAPI **58**, to provide file encryption keys, and generate decryption field information.

The EFS service **50** also provides support for Win32 APIs **32**, which are programming interfaces for encrypt, decrypt, recover and provide support for importing and exporting encrypted files. Importing and exporting encrypted files allows users to convert the files into opaque data (encrypted) for operations such as backup, restore, and general file transfer purposes as described below. The Win32 APIs **32** provide programming interfaces for encrypting plain text files, decrypting or recovering ciphertext files, and importing and exporting encrypted files (without decrypting them first). These APIs **32** are supported in a standard system DLL, advapi32.dll.

The EFS service **50** provides a number of services, including generating a session key and exchanging it with the EFS driver **46** and the FSRTL **48**. Based on the EFS driver **46** request, the EFS service **50** generates a cryptographically strong session key (using CryptoAPI) and communicates it to the driver and FSRTL **48**. The EFS service **50** also generates the file encryption keys in fields stored with the file (the Data Decryption Field, or DDF, and the Data Recovery Field, or DRF, described below with reference to FIGS. **3–5**) using the user's and recovery agents' public key defined for EFS. When the EFS driver **46** requests a new file encryption key, the EFS service **50** generates this information using CryptoAPI and returns it to the EFS driver **46**.

The EFS service **50** also extracts the file encryption key, i.e., when the EFS driver **46** requests this operation, the EFS driver **46** supplies the file metadata, including the DDF and DRF key fields. Based on that information, the EFS service **50** sequentially searches the DDF and (if necessary) the DRF key fields to extract the name of the user's key therefrom, and accesses its private portion via the CryptoAPI provider **58**. If successful, (as described in more detail below), it passes the encrypted file encryption key to the provider for decryption. The service verifies that the decryption was correct (as also described below), and also verifies that the keys used in the file metadata are up to date. If the keys are not up to date, the service regenerates the metadata (DDF and/or DRF) and returns the extracted file encryption key and the metadata back to the EFS driver **46**.

When the EFS driver **46** is loaded by NTFS **28**, it first initializes its structures, and reserves some space to ensure that some memory is always available thereto. Then, the EFS driver **46** registers itself with NTFS **28**. Lastly, to synchronize with the driver, the EFS driver **46** attempts to create a new event. If the event is successfully created, this indicates that the EFS service **50** has not been initialized and the EFS driver **46** has been loaded first. If successful, the EFS driver **46** then creates a thread waiting on the event to be signaled. Later, when the event is signaled, i.e., the EFS service **50** is ready to communicate, the EFS driver **46** calls the EFS service **50** to get the session key. Once the session key has been transferred from the EFS service **50** to the EFS driver **46**, and the EFS service **50** and the EFS driver **46** are synchronized. Note that if the event was not successfully created, it is ordinarily because the EFS service **50** was already initialized, in which event the EFS driver **46** simply makes the call to get the session key.

In the situation where the EFS service **50** was loaded first, the EFS service **50** tries to create a new event. If the event is successfully created, then the EFS driver **46** has not been initialized. The EFS service **50** generates the session key

without waiting on the event. Later, when the EFS driver **46** is loaded, the EFS service **50** will be called by the EFS driver **46** to provide the session key thereto. When the EFS service **50** provides the session key, the EFS service **50** closes the event which was created earlier by the EFS service **50**, and the EFS service **50** and the EFS driver **46** are synchronized. Note that if the event was not successfully created, it is ordinarily because the EFS driver **46** was already initialized, in which event the EFS service **50** instead opens the event and signals the event to let the EFS driver **46** know that the EFS service **50** is ready. Thereafter the EFS service **50** is asked for the session key by the EFS driver **46**, and the synchronization is done.

WIN32 SYSTEM APIS

As described in more detail below with particular respect to FIGS. **22–23**, the EFS service **50** also provides a number of other user mode interfaces. These interfaces work closely with WIN32 system APIs to enable the user to perform operations such as convert an existing plaintext file to an encrypted file, convert an encrypted file to a plaintext file, and provide a backup and restore mechanism. By way of example, the Win32 interfaces **32** work with the EFS service **50** to expose EFS functionality, and include EncryptFile, which is a wrapper that calls into the interface provided by the EFS service **50** to do file encryption (FIG. **22**). Another interface, DecryptFile, is a wrapper that similarly calls into the interface provided by EFS service **50** to do file decryption/recovery (FIG. **23**).

A Backup/Restore mechanism is also provided in the system APIs, which enables users and backup operators to backup encrypted files without decryption. To this end, an OpenRawFile interface allows the user to open an encrypted file without read access, and without setting up a file encryption key to do transparent reads and writes. For these operations, NTFS **28** recognizes the access level and does not call the encryption driver **46** to look up a key for this file, nor to decrypt reads nor encrypt writes. The only operations allowed on a file opened via this interface are file controls. Thus, a ReadRawFile interface allows the user to read all the data from the file, including the encryption metadata, as a contiguous opaque stream that can be backed up and later restored. A WriteRawFile interface allows the user to write all the data to the file from its backup, including the encryption metadata, to re-create the encrypted file. Lastly, a CloseRawFile is provided that allows the user to close the file which was opened raw by OpenRawFile.

A FileControl interface allows the Win32 APIs that provide the backup and restore mechanism to read and write raw encrypted data. Note that such raw data reads and writes are from/to NTFS **28** direct to/from the disk (storage); EFS becomes involved because the EFS service **50** and the EFS driver **46** share the common session key, and all file controls need to be verified (as described below). For backing up the file, the Win32 APIs read the EFS metadata via an EFS file control, which translates into the FileSystemControl_2 that returns the EFS stream. Then, NTFS **28** file controls are called to read the actual file data, which is packaged into an opaque stream and written out. To (gradually) restore the file, the reverse process is performed. An EFS file control is called to identify a first stream and another EFS file control to write the raw data back.

DATA ENCRYPTION

As conceptually represented in FIGS. **3–5**, the present invention implements data encryption and decryption using a public key-based scheme. To this end, file data is encrypted using a fast symmetric algorithm with a file encryption key (FEK) **60** (FIG. **3**). The FEK **60** is a randomly generated key of a certain length required by the selected algorithm, or as otherwise required if the algorithm supports variable length keys. As represented in FIG. **3**, a random number generator **62** generates the FEK **60**. To encrypt the file data using the FEK **60**, the plain text **64** of the file is encrypted by a file encryption mechanism **66** using an appropriate algorithm (e.g., DES) and written as encrypted text **68** to an encrypted file **70**.

In accordance with another aspect of the present invention, and as shown in FIG. **3**, the randomly generated FEK **60** is itself encrypted with the public key **72** of at least one user, and stored with the encrypted file **70** in a special EFS attribute called the Data Decryption Field (DDF) **74**. Using a suitable encryption algorithm, (e.g., RSA), a data decryption field generator **76** performs the key encryption. In keeping with public-key based schemes, the private portion of the user's key pair is only used during decryption, i.e., an encrypted FEK **60** in the data decryption field **74** is decrypted using the private portion of the key pair. The private portion **84** (FIG. **4**) of a user key pair is safely stored in a separate location, such as on a smart card and/or other secure storage device. Note that encryption can also be done using a symmetric algorithm, such as a password-derived key, but while feasible, EFS preferably does not support such encryption because password-based schemes are inherently weak due to dictionary attacks and the like.

In accordance with one aspect of the present invention, as also represented in FIG. **3**, the FEK **60** is also encrypted using one or more recovery public keys **78**. The recovery key encryption public keys **78** belong to trusted persons, known as recovery agents, as specified by a recovery policy, described below. Similar to the FEK's encryption using the user's public key, the public portion of each recovery key pair is used to encrypt the FEK **60** using a data recovery field generator **80**, (employing, for example, a suitable encryption algorithm, such as RSA, which need not be the same algorithm used to encrypt the FEK **60** with the user's public key). This list of encrypted FEKs is similarly stored along with the file **70** in a special EFS attribute called the Data Recovery Field (DRF) **82**. Thus, only public portions of the recovery key pairs are needed for encryption of the FEK **60** in the DRF **82**. Note that to facilitate proper operation, these public recovery keys are to be present at all times on an EFS system for normal file system **28** operations, since a user may wish to encrypt a file at any time. Recovery itself is expected to be a rare operation required only when users leave organizations, lose keys, and so on. As a result, recovery agents are also able to store the private portions **90** (FIG. **5**) of the keys on smart cards, floppy disks, and/or other secure storage devices.

In keeping with the invention, the Encrypting File System architecture is not limited to any particular encryption algorithm, but rather is fully algorithm agile and may use any cryptography algorithm for the various encryption phases. As a result, the Encrypting File System of the present invention allows for the usage of better and better encryption algorithms as such algorithms advance technologically. Moreover, the user is able to choose from among available algorithms to select an algorithm having greater or less security (i.e., based on how sensitive the user thinks the information is) versus the speed of encryption and decryption, (i.e., more secure algorithms are generally slower). Thus, in the above description, DES is one such algorithm used to encrypt file data, while RSA is used to encrypt the FEK.

DATA DECRYPTION

FIG. 4 conceptually shows the user decryption process. A user's private key **84** (or a recovery agent's private key **90**) is used to decrypt the corresponding encrypted FEK item stored in the data decryption field **74**. To accomplish the decryption of the key, each encrypted FEK item in the DDF **74** (and, if necessary the DRF **82**) and the user's or recovery agent's private key are iteratively fed into an extraction mechanism **86** (in the EFS service **50**), until a match is found which properly decrypts the FEK **60**. From there, the FEK **60** is fed into a file decryption mechanism **88** which uses an appropriately corresponding decryption algorithm in a known manner to decrypt the encrypted text **68** into plain text **64**. Note that when multiple decryption algorithms are available, the data decryption field may store information about the encryption algorithm so that the file decryption mechanism **88** uses the correct decryption algorithm. Only one encryption algorithm can be used per file, although each of several files can have its own such algorithm.

While a file is open, the decrypted FEK **60** is saved by the file system **28** in association with the file **70**. As shown in FIG. **6**, with NTFS **28**, stream control blocks **94** maintain file information for each open file, and each stream control block (e.g., 94_1) corresponding to an encrypted file has a key context (e.g., 96_1) pointed to thereby. The key context **96** maintains the information necessary to encrypt and decrypt a file during writes and reads to the disk, respectively. As described in more detail below, the FEK **60** is used to decrypt file data reads on a block-by-block basis, i.e., random access to a large file will decrypt only the specific blocks read from disk for that file. The entire file is not necessarily decrypted.

FILE RECOVERY

FIG. **5** conceptually illustrates the recovery process. The recovery process is similar to user decryption, except that the process uses a recovery agent's private key **90** to decrypt the FEK **60** in the DRF **82**. Consequently, no match will be found in the DDF **74** and thus the search for a match will continue into the DRF **82**. To initiate a recovery, the recovery agent submits his or her private key **90**, and a data recovery field extraction mechanism **92** (which is preferably the same data decryption field extraction mechanism **86** of FIG. **4** described above) iteratively uses the agent's private key **90** to search the DDF **74**. Then, since it is a recovery agent and not a user, no match is found in the DDF **74** and thus the extraction mechanism continues to search the DRF **82**.

Regardless of whether it is a normal user opening or an opening for a recovery, once a key is found that is in the current context, (CryptoAPI **58** maintains a set of keys for each user), then the key is verified by comparing it to known information decrypted with that key. More particularly, at the time of encryption, the user's public key is appended to the FEK of the file, which is then encrypted with the FEK **60** as known information. If the found key decrypts the stored information to equal the known information, then the key is verified. This scheme provides a strong encryption technology as it provides one of many possible recovery agents with the ability to recover the file, thereby providing organizations with redundancy and flexibility in implementing recovery procedures.

EFS thus provides a built-in data recovery support, referred to as the "Recovery Policy". The preferred system enforces configuration of recovery keys, and is intentionally limited to only being usable when the system is configured with one or more recovery keys. The file recovery operation only divulges the randomly generated file encryption key **60**, and not the user's or any other recovery agent's private key. As can be appreciated, this is ideal for most business environments where an organization may need to recover data that was encrypted by an employee after an employee leaves the organization or loses his or her key.

The recovery policy, also known as the EFS policy, may be defined at the domain controller of a Windows NT Domain, whereby the policy is enforced at all machines in that domain. The policy contains the public keys of the recovery agents. As a result, the recovery policy is only under the control of the domain administrators, thereby providing controls on who can recover the data. To enable the use of encryption features on a standalone Windows NT workstation in a home environment, as an added feature, EFS automatically generates recovery keys and saves them as machine keys, thereby reducing the administrative overhead for an average user.

In a domain, the recovery policy is sent to each of the machines on the domain, whereby even when not connected, the local machine maintains the policy therewith in a local security authority, LSA. In this manner, EFS can operate to encrypt files when not connected. Each time a machine joins the domain, or if the recovery policy changes, the policy is propagated to the machines in the domain when they are connected. Moreover, every time a file is opened, the metadata for that file is compared against the recovery policy to see if the recovery policy has changed, and if so, the metadata is updated with the FEK encrypted with the new user and/or recovery agent public key information. Safeguards, including hashes (MD5), are used to ensure that the recovery policy is not changed, such as by a malicious user.

For example, a malicious user may wish to make a file unrecoverable by changing the DRF **82**. To detect this, using a section of the DRF **82**, a crytopgraphy hash (MDS) is created, signed with the FEK **60** of the file, and stored with the DRF **82**. Later, when the file is opened and the FEK **60** is obtained, the section is decrypted with the FEK **60** to see if the information stored in the DRF **82** matches. If so, the stored recovery policy is proper, otherwise the recovery policy is replaced with the current recovery policy of the domain.

If a machine is not part of a domain, it still has a local recovery policy, but the recover keys are generated by and kept on the machine as describe above. In this manner, every file encrypted under EFS always has some recovery policy. If the machine later becomes part of a domain, the local recovery policy is wiped out and replaced with the domain policy.

GENERAL OPERATION

Turning to an explanation of the operation of the invention and beginning with the flow diagram of FIG. **10**, when a application **30** wishes to create or open an encrypted file, the application **30** first calls an appropriate API **32** requesting a new file be created in an encrypted directory, or an encrypted file be opened.

As shown in FIG. **10**, once a create or open request is received, the I/O subsystem **56** arranges for passing the request as an IRP to the appropriate file system, e.g., NTFS **28**, at steps **1000–1002**. However, as described above, the IRP is first received by the EFS driver **46** at step **1100** of FIG. **11**, which recognizes the IRP as specifying an open/create operation.

The EFS driver **46** begins the EFSCreateFile operation by performing some preprocessing as shown in FIG. **11**. As

represented by step **1102** of FIG. **11**, the EFS driver **46** allocates an EFS Context Block 98_1 for this file, and adds it to the EFS context block chain **98** (FIG. **7**). Note that the EFS context block 98_1 is created for each new file, even though the file may not be encrypted, because the EFS driver **46** does not know at this time whether the file is already encrypted or is to be encrypted. The EFS context block 98_1 includes status information, which is initialized with "No processing needed," an IRP pointer pointing to the current file object, and an EFS metadata stream initialized to NULL. Lastly, at step **1104**, the IRP is passed to NTFS **28**.

As shown in step **1006** of FIG. **10**, when NTFS **28** receives the IRP from the EFS driver **46** and recognizes it as an NTFS **28** Create packet, NTFS **28** handles the create/open IRP. FIG. **12** generally shows how NTFS **28** handles the IRP. First, as represented by step **1200**, information in the IRP is tested to determine if an existing stream is to be opened, or a new stream on file/directory is to be created. If an existing stream is to be opened, NTFS **28** opens the stream at step **1202**. At step **1204**, NTFS **28** determines if the file or its directory is encrypted, and if so, at step **1208** calls the FSRTL **48** open/create callout, described below with reference to FIG. **13**.

Conversely, if a new stream was created as determined by step **1200**, NTFS **28** next determines at step **1206** if the parent directory is encrypted. If the parent is encrypted, at step **1208**, NTFS **28** calls the FSRTL **48** open/create callout, as described below. Note that if neither the parent is determined to be encrypted (step **1206**) nor the file or directory encrypted (step **1204**), NTFS **28** does not make the FSRTL **48** callout, whereby NTFS **28** simply performs any tasks it needs to at step **1210** before returning to the EFS driver **46**.

The steps of the EFS create/open FSRTL **48** callout are generally represented in FIG. **13**, wherein the FSRTL **48** callout begins by first examining the type of access requested by the user (step **1300**). If an existing stream on the file or directory is opened without read, write, append or execute (R/W/A/E) access, the call is simply succeeded, as no encryption/decryption is needed, e.g., the user wants to read attributes, and attributes are not encrypted. Otherwise, at step **1302**, the FSRTL **48** searches the EFS context chain **98** for the appropriate file object corresponding to this file (allocated at step **1102** of FIG. **11**). The Create/Open callout performs operations based on the type of file/directory, as set forth below.

If the type of file is an existing file (step **1304**), then FSRTL **48** was called because either a new stream was created or an existing stream was opened. If so, the user needs to be verified, and the callout process continues to step **1400** of FIG. **14**. At step **1400**, the EFS metadata from the file is read using an (NtOfs) API. Then, at step **1402**, the metadata that was read is set up in the context block 98_1 and the status on the block changed to indicate "User Verification Required." Then the key context **96** is checked at step **1404**, and if the NTFS key context **96** is NULL, then a key is needed. If the key context **96** is NULL (step **1404**), this file was not read, and thus there is no possibility that decrypted file data is present in the cache memory. As a result, the context block is set to indicate "No Cache Check Needed" at step **1406**. Lastly, if a new stream was created as determined by step **1408**, the context block 98_1 is set to indicate "Turn On Encryption Bit" at step **1410**.

If instead the type of file is a new file (step **1306** of FIG. **13**), a new FEK and EFS metadata are needed. First, at step **1308**, the EFS metadata is read from the parent directory using NtOfs API. Step **1310** sets up the metadata that was just read in the context block 98_1, and changes the status on the block to "New File Efs Required," "Turn On The Encryption Bit" (step **1312**) and "No Cache Check Needed (step **1314**)."

If instead the file object type indicates a new directory (step **1320**), only new EFS metadata is needed. There is no FEK in this case, because at present, streams in the directory are not encrypted. Accordingly, the EFS metadata from the parent directory is read at step **1322** (using NtOfs API). Then, at step **1324**, the metadata that was just read is set up in the context block 98_1 and the status on the block changed to "New Directory Efs Required," "No Cache Check Needed" (step **1326**) and "Turn On The Encryption Bit" (step **1328**).

Lastly, if the type represents an existing directory (step **1332**), either a new stream was created or an existing stream was opened. At present, no action is taken because the directory data streams are not encrypted. However, it can be readily appreciated that directory streams also may be encrypted in the same manner that file data streams are encrypted using the encrypting file system of the present invention.

As shown in FIG. **12**, step **1210**, the callout returns to NTFS **28**, whereby NTFS **28** can perform any of its own operations. The file/create process returns to step **1010** (FIG. **10**) the EFS filter driver **46** for post-processing.

The EFS Create/Open File post-processing process is represented in FIGS. **15–19**. Beginning at step **1500** of FIG. **15**, the context block 98_1 is evaluated for "No Cache Check Required" status. If a cache check is required, the process branches to step **1502** where the caller's security context along with the EFS ID for the file stored in the EFS stream are used by the EFS cache to check if this file was successfully opened by the user the recent past. If so, the call is succeeded since the cache already contains the appropriate information.

If not in the cache, step **1504** checks if read data, write data, append data or execute access is requested. If none of these are requested, but a new stream was created as determined by step **1506**, and the context block 98_1 indicates "Turn On Encryption Bit" status (STEP **1508**), the EFS data stream is not needed and is released. Only the encryption bit on the stream needs to be turned on, which is performed at step **1510**. The post processing is complete and the overall process returns to step **1012** of FIG. **10**.

However, if none of the situations identified above with respect to FIG. **15** are satisfied, different operations need to be performed based on the status information, which is tested beginning at step **1600** of FIG. **16**. First, if the status in the context block 90_1 indicates that user verification is required at step **1600**, the EFS driver **46** impersonates the security context (provided in the IRP) at step **1602**, and at step **1604** calls the EFS service **50**, passing the EFS metadata to request the FEK.

At step **1606**, the EFS service **50** responds to the call by impersonating the context, and using information in the EFS metadata, looks up the user's private key to decrypt the FEK. The EFS service **50** may also update the EFS metadata (step **1610**) if the user's key has been updated or any recovery keys are updated as determined by step **1608**. In any event, at step **1612**, the EFS service **50** verifies the integrity of the FEK **60** and returns all information back to the EFS driver **46**. More particularly, to verify integrity, a key integrity block is constructed as follows:

[F(FEK,Puk),FEK]Puk

where

F( ) is a suitable hash function (preferably MD5),

Puk is the user's public key, and

[ ]Puk denotes encryption with the user's public key

Consequently, when it is believed that a valid FEK has been decrypted with a user's public key, the block above is computed with the present information and compared to the block stored on the file. If they match, the key integrity is verified.

Alternatively, if the status did not indicate that user verification was required (step **1600**), but instead indicated that a new file FEK was required, step **1614** branches to step **1700** of FIG. **17**. At step **1700**, the EFS driver **46** impersonates the securing context in the IRP, and at step **1702** calls the EFS service **50** passing the parent directory's EFS metadata, requesting a new FEK and EFS metadata. At step **1704**, the EFS service **50** impersonates the context and generates a random FEK **60**. If the user does not have a key as determined by step **1706**, at step **1708**, the EFS service **50** auto-generates a key pair for the user. Lastly, step **1710** creates the EFS metadata stream with the FEK **60** encrypted under the user's public key and the recovery agent's public keys. The EFS service **50** also encrypts the FEK **60** using all the public keys in the parent directory's EFS metadata so that users who are allowed access to the parent directory also have access to the file (provided NTFS **28** access control lists allow such access).

If neither step **1600** nor **1614** was satisfied, the post-process branches to step **1616** to determine if the EFS context indicated that a new directory FEK is required. If so, the post-process branches to step **1800** of FIG. **18** wherein the EFS driver **46** impersonates the securing context in the IRP. Step **1802** calls the EFS service **50**, passing the parent directory's EFS metadata and requesting new EFs metadata. Note that no FEK is needed, as directory streams are not encrypted at this time. However, in the future, directory streams will also encrypted in the same manner that file streams are encrypted in accordance with the present invention.

In any event, at step **1804**, the EFS service **50** impersonates the context, and, using an empty FEK, creates the EFS metadata stream. Step **1706** checks to see if the user does not have a key, and if not, at step **1708**, the EFS service **50** auto-generates a key pair for the user. Then, at step **1810**, the empty FEK is encrypted under the user's public key and the recovery agent's public keys, and the FEK is also encrypted using all the public keys in the parent directory's EFS metadata so that users allowed access to the parent directory also have access to the file if the access control lists allow such access.

Ultimately, regardless of which of the three statuses were in the context, the post-process winds up at step **1900** of FIG. **19** to issue an appropriate FSCTL. Two such FSCTL calls are available, FSCTL_SET_ENCYRPTION and FSCTL_ENCRYPTION_FSCTL IO. FSCTL_SET_ENCYRPTION tells NTFS **28** to turn on or turn off the encryption bit for a stream. The FSCTL_ENCRYPTION_FSCTL IO is a miscellaneous FSCTL used for performing a number of operations, described below.

To this end, the FSCTLs are accompanied by a data structure **100**, as shown in FIGS. **8** and **9**. The data structure includes a public code so that NTFS **28** can differentiate between the two types of FSCTL calls, along with an EFS subcode to more particularly define the operation for the EFS driver **46** and/or the EFS service **50**. The data structure also includes EFS data specifying either FEK information (FIG. **8**) or file handle information (FIG. **9**), and, at times, EFS metadata. For purposes of security, the EFS subcode and EFS data fields are encrypted with the session key established when the EFS service **50** is initialized, as described above.

In use, the FSCTL_SET_ENCRYPTION may be issued, for example, to turn on the encryption bit for a file when that file is first put into an encrypted directory. The subcode indicates whether the encryption bit should be turned on or off. For such an operation, the FEK **60** is already known, and thus as shown in FIG. **8**, the EFS data includes the FEK, and the FEK encrypted with the session key. Note that all but the public code is encrypted with the session code. To verify the integrity and the source of the data structure **100**, the encrypted portion of the data structure is decrypted. Then, the encrypted FEK is decrypted and compared with the other FEK, and if equal, the structure is verified. The EFS stream, if available may also be compared with the EFS metadata, if otherwise known. Since the FEK is not always known, a similar verification is performed using the session key and the file handle as shown in FIG. **9**. A repeated file handle is actually used in the appropriate fields (FIG. **9**) so as to equal a length of eight bytes.

The callouts to the FSRTL **48** (via FileSystemControl_1 or FileSystemControl_2, and passed through NTFS **28**) are used to overwrite attributes or set attributes depending on an accompanying subcode. Two bits of the subcode represent the overwrite attributes or set attributes operations. Note that when the EFS stream is to be written for a new file, FILE_SYSTEM_CONTROL_1 is used with the FSRTL **48** callout to also turn on the encryption bit (as performed by NTFS **28**, described above). Alternatively, FILE_SYSTEM_CONTROL_2 is used with the callout when no change to the encryption bit is needed, for example, if the user has simply changed user keys.

Regardless which is used, one bit of the subcode represents the operation "Set EFS KeyBlob," which indicates to the FSRTL **48** that new encryption key information is available and needs to be entered into the appropriate key context. Another bit represents the operation "Write EFS Stream." Write EFS Stream is issued by the EFS service **50**, such as when the user or recovery agent has changed a public key and the file metadata needs to be rewritten with the EFS metadata in the data structure **100**. One other subcode represents "Get EFS stream," which results in the current EFS attributes for a file being written into the EFS field, such as when a user wants to export a stream or wants to know a key name.

Thus, returning to FIG. **19**, step **1900** tests if the only requirement is to turn on the encryption bit, and if so, issues the FSCTL_SET_ENCYRPTION control at step **1902** with the subcode indicating that the bit should be turned on. Of course, the other EFS data including the session key, handle, handle and encrypted copy of same is also in the data structure **100** for verification purposes. In any event, the FSCTL reaches NTFS **28**, which turns around and sets the encryption bit on the stream (and on the file if not already on) and calls an FSRTL callout to pass the information to the FSRTL **48**.

If step **1900** is not satisfied, the ENCRYPTION_FSCTL_IO FSCTL needs to be called with an appropriate subcode in the data structure **100**. Thus, if the status is "User Verification Required" (step **1904**), the subcode is set to KeyBlob at step **1905**. Next, if the metadata has been updated as determined by step **1906**, step **1907** sets the Write EFS Stream subcode bit before the call at step **1914**. Otherwise, if the status is "New File FEK Required" (step **1908**), the subcode is set to KeyBlob and Write EFS Stream

at step **1910**, i.e., both bits are set. If neither of these, then the status is "New Directory FEK Required," and the subcode is set to Write EFS Stream at step **1912**, i.e., only the other bit is set. The FSCTL is issued at step **1914**.

READ AND WRITE

Turning to an explanation of the read and write operations of the present invention, as first shown in FIG. **20**, when an application **30** requests to read some data from the open file (step **2000**), the I/O subsystem **56** receives the read request and passes it as an IRP to the appropriate file system **28**, e.g., NTFS. First, however, the IRP is received by the EFS driver **46**, which recognizes the IRP as corresponding to a read request, and as a result, directly hands the IRP to NTFS **28** at step **2002**. At step **2004**, NTFS **28** reads the encrypted data from disk into a buffer just as it would read the plaintext data for any other file. However, for the encrypted file, the file system **28** recognizes at step **2006** that this file is encrypted, and at step **2008** remembers and gets the key context 96_1 that the encryption driver **46** earlier had returned from the create/open callback. At step **2010**, NTFS **28** uses the AfterReadProcess callback and provides the registered function with the data and enough information, including the encryption context, to decrypt the data. In general, the information includes the offset into the file, a pointer to the read buffer, the length to read, and the key. At step **2012**, the encryption driver **46** decrypts the data and returns it to the file system **28**, whereby at step **2014** the file system **28** then returns this plaintext through the I/O subsystem **56** to the application in the normal way. Note that however that certain NTFS **28** internal metadata streams containing file indexing and other such information are not encrypted. NTFS **28** recognizes these streams at step **2006**, and temporarily skips over steps **2008–2012** for these particular streams.

As shown in FIG. **21**, when an application **30** requests to write data to the open file (step **2100**), the I/O subsystem **56** receives the write request and passes it as an IRP to the appropriate file system **28**, e.g., NTFS. First, however, the IRP is received by the EFS driver **46**, which recognizes the IRP as corresponding to a write request, and as a result, directly hands the IRP to NTFS **28** at step **2102**. At step **2104**, NTFS **28** copies the write data into a separate buffer so that no changes can be made to the data that is to be written. For the encrypted file, the file system **28** recognizes at step **2106** that this file is encrypted, and at step **2108** remembers and gets the key context 96_1 that the encryption driver **46** earlier had returned from the create/open callback. At step **2110**, NTFS **28** uses the BeforeWriteProcess callback and provides the function with the data and enough information, including the encryption context, to encrypt the data. At step **2112**, the encryption driver **46** encrypts the data and returns it to NTFS **28**, whereby at step **2114** the file system **28** then writes the now-encrypted data in the separate buffer to the non-volatile storage **40** in the normal way, i.e., as if it was plaintext data for any other file. Again note that the NTFS **28** internal metadata streams containing the file indexing and other such information are not to be encrypted. NTFS **28** recognizes these streams at step **2106**, and temporarily skips over steps **2108–2112** for these particular streams.

ENCRYPT AND DECRYPT FILE APIs

EFS also provides APIS **32** to facilitate encryption and decryption of stored files. The Win32 EncryptFile API is used to encrypt a plaintext file/directory. AS shown in FIG. **22**, with this API, the application **30** (user) provides the name of the file to encrypt at step **2200**, and this call translates into a call to the EFS service **50** to do the operation. At step **2202**, the EFS service **50** opens the file on the user's behalf, makes a backup copy for crash recovery purposes and at step **2204** marks it for encryption by issuing the SET_ENCRYPT file control (FSCTL). At step **2206**, the EFS service **50** then reads data from each stream in the copy and writes the data back to the original file at step **2208**. Note that during the write operation, because the encryption bit is set, the data is automatically encrypted before being written to the disk. The process is repeated via step **2210** until all data streams are written. If this call completes successfully (step **2212**), the backup is deleted at step **2214**, otherwise the original file is restored at step **2216** and the call is failed. In the case of a directory, the directory is simply marked encrypted, as there is no data to encrypt. Note that, as described above, NTFS **28** internal metadata streams are not encrypted.

A WIN32 DecryptFile API is also provided by the EFS service **50**, and is the converse operation of the encrypt file/directory operation. As shown in FIG. **23**, at steps **2300–2302**, the EFS service **50** is provided with the file name and opens the file on the user's behalf. Steps **2306–2310** read the data from all streams and write those streams into a copy, which is plaintext, as decryption happens transparently. At step **2312**, the service then issues the decrypt file control to delete the metadata and remove the encryption attribute. Then, as shown by steps **2314–2318**, the API writes back all the data streams from the copy over the original, which are written in plaintext. If this completes successfully, the copy is deleted at step **2322**, otherwise the original is restored at step **2324**. In the case of a directory, the directory is simply marked as decrypted to delete the metadata and the attribute, as there is no data to decrypt.

As can be appreciated, EFS file encryption is supported on a per file or entire directory basis (although NTFS **28** operates per stream). Directory encryption is transparently enforced, i.e., all files (or subdirectories) created in a directory marked for encryption are automatically created encrypted. Moreover, file encryption keys are per file, making them safe against move/copy operations on the file system volume. Unlike existing application-level schemes, the file need not be decrypted before use, since, as will become apparent below, the encryption and decryption operations will be done transparently and on the fly when bytes travel to and from the disk. EFS will automatically detect the encrypted file and locate the user's key from a key store. The mechanisms of key storage are leveraged from CryptoAPI, and as a result the users will have the flexibility of storing keys on secure devices such as smart cards and/or floppy disks.

Moreover, in keeping with the invention, EFS cooperates with the underlying file system **28** (e.g., NTFS), whereby EFS facilitates the writing (by a properly developed application program) of encrypted temporary files. With such an application program, when temporary files are created, the attributes from the original file are copied to the temporary file making the temporary copy also encrypted. In addition, the EFS driver **46** is a Windows NT kernel mode driver, which uses the non-paged pool to store file encryption key, thereby ensuring that the key never makes it to the page file.

The EFS architecture allows file sharing between any number of people by simple use of the public keys of those people. Each user can then independently decrypt the file using their private keys. Users can be easily added (if they have a configured public key pair) or removed from the clique of sharers.

In a stand-alone configuration, EFS allows users to start encrypting/decrypting files with no administrative effort to

set up a key, i.e., EFS supports auto-generation of a key for the user if one is not configured. With a domain configuration, an administrator only needs to set up a domain policy once for EFS to become operational. Lastly, EFS will also support encryption/decryption on remote files stored on file servers. However, the data once read from the disk is decrypted on the fly and hence may travel in plaintext on the wire if the file sharing protocol does not implement communication encryption, i.e., EFS addresses storage encryption, not communication encryption. Communication protocols can be used to provide such encryption.

As can be seen from the foregoing detailed description, there is provided a system and method for encrypting data, the system and method integrated into a file system such that encryption and decryption work transparently to legitimate users. The system and method provide for the ability to share sensitive data among more than one legitimate user, and adding and removing access for a user is simple. A strong cryptographic solution is provided that addresses encrypted data recovery, such as when users lose keys. The system and method are flexible and extensible.

While the invention is susceptible to various modifications and alternative constructions, a certain illustrated embodiment thereof is shown in the drawings and has been described above in detail. It should be understood, however, that there is no intention to limit the invention to the specific form disclosed, but on the contrary, the intention is to cover all modifications, alternative constructions, and equivalents falling within the spirit and scope of the invention.

What is claimed is:

1. In a computer system having a file system, a method of encrypting or decrypting data in a file stored in a non-volatile storage, comprising:

receiving information at the file system indicating that the file is designated as encrypted;

receiving an encryption key associated with the file;

receiving a request to write file data to non-volatile storage and receiving the file data, and in response, encrypting the file data into encrypted file data at file system level software using the encryption key, writing the encrypted file data to non-volatile storage and writing encryption key information in association with the file to the same non-volatile storage as the encrypted file data; and

receiving a request to read file data from non-volatile storage, and in response, reading the encrypted file data from the non-volatile storage, decrypting the encrypted file data into decrypted file data at the file system level software using the encryption key, and returning the decrypted file data.

2. The method of claim 1 wherein the encryption key is encrypted with at least one public key before writing the encryption key information to the non-volatile storage in association with the file.

3. The method of claim 2 further comprising obtaining the encryption key via a private key.

4. The method of claim 2 wherein the encryption key is encrypted with a plurality of public keys into a plurality of encrypted keys and each of the encrypted keys are stored with the file.

5. The method of claim 4 wherein at least one encrypted key is encrypted with the public key of a user and at least one other encrypted key is encrypted with the public key of a recovery agent.

6. The method of claim 5 wherein the file encryption keys encrypted with the public key of each user is stored in a separate field from the file encryption keys encrypted with the public key of each recovery agent.

7. The method of claim 6 further comprising verifying the integrity of the encryption key information stored with the file.

8. The method of claim 7 wherein verifying the file encryption key includes hashing the file encryption key with the public key of a user, encrypting the hashed file encryption key and the file encryption key with the public key of the user, and storing the hashed file encryption key and the file encryption key with the file.

9. The method of claim 5 further comprising receiving a private key of either a user or a recovery agent, and scanning the encrypted file encryption keys stored with the file by using the private key until a match is detected.

10. The method of claim 1 wherein a recovery policy is stored in association with the file, and further comprising verifying the integrity of the recovery policy.

11. The method of claim 10 wherein the recovery policy is stored in a field of the file having recovery agent information therein, and wherein verifying the integrity of the recovery policy includes hashing at least part of said field, encrypting the hashed part with the file encryption key and storing the encrypted hashed part as information with said field, retrieving the information and comparing the information with current information.

12. The method of claim 1 further comprising generating a random number and basing the file encryption key thereon, and wherein the file encryption key is encrypted with at least one public key and stored in association with the file.

13. The method of claim 1 wherein encrypting and decrypting data each include placing a callout to a run-time library of software functions.

14. In a computer system having a file system, a method of reading unencrypted file data or encrypted file data and returning the data read as unencrypted file data, comprising, receiving at the file system from a requesting program a request to read file data from a non-volatile storage, reading the file data, determining at file system software if the file data is encrypted, and if the file data is not encrypted, returning the file data to the requesting program, and if the file data is encrypted, obtaining a file encryption key for that file by applying a private key to the file encryption key data, the file encryption key data including the file encryption key encrypted with a public key and stored on the same non-volatile storage and in association with the file, providing the file encryption key and the file data to a file system level decryption mechanism, decrypting the file data into unencrypted file data, and returning the unencrypted file data to the requesting program.

15. The method of claim 14 wherein determining if the file data is encrypted includes checking an encryption bit stored in association with the file data.

16. The method of claim 14 wherein receiving a request to read file data from the non-volatile storage includes notifying the file system of a raw data read, wherein encrypted data is read from the non-volatile storage and returned without providing said data to the decryption mechanism.

17. The method of claim 14 further comprising verifying the file encryption key.

18. The method of claim 17 wherein verifying the file encryption key includes decrypting encrypted file encryption key information stored with the file.

19. The method of claim 14 wherein the file encryption key is separately encrypted with a plurality of public keys into a plurality of sets of file encryption key data, at least one public key being of a user and another being of a recovery agent, and wherein each set of encrypted file encryption key data is stored in association with the file.

20. The method of claim 19 further comprising receiving a private key of either a user or a recovery agent, and scanning the encrypted file encryption key data stored with the file by applying a private key to the encrypted file encryption key data until known information stored with the file encryption key data is detected.

21. The method of claim 19 further comprising verifying the integrity of information stored with respect to at least one recovery agent.

22. In a computer system having a file system, a method of storing selected file data as encrypted file data, including receiving at the file system a request to write the file data and the file data to be written as encrypted, encrypting the file data via file system level software into encrypted file data by using a file encryption key, encrypting the file encryption key with a public key, writing the encrypted file data to a non-volatile storage, and writing the encrypted file encryption key to the same non-volatile storage as the encrypted file data and in association therewith.

23. The method of claim 22 wherein receiving a request to write file data to the non-volatile storage includes notifying the file system of a raw data write, wherein data is written to the non-volatile storage without further encrypting said data.

24. In a computer system having a file system, a system for encrypting data written by the file system to a non-volatile storage, comprising, means for obtaining a file encryption key, a software encryption mechanism at a file system software level for converting unencrypted data to encrypted data based on the file encryption key, the file system writing at least some of the data as encrypted data to a file in the non-volatile storage, and means for encrypting the file encryption key, the file system writing the encrypted file encryption key to the same nonvolatile storage as the encrypted data and in association therewith.

25. The system of claim 24 further comprising means for retrieving the file encryption key stored in association with the file, including means for decrypting the encrypted file encryption key, and a software decryption mechanism at the file system software level for converting encrypted data to decrypted data based on the file encryption key.

26. The system of claim 24 wherein the means for obtaining a file encryption key includes a random number generator.

27. The system of claim 24 wherein the means for obtaining a file encryption key includes an interface to a cryptography service.

28. The system of claim 25 wherein the means for encrypting the file encryption key includes means for accessing a public key of at least one user.

29. The system of claim 28 wherein the means for decrypting the file encryption key includes means for receiving a private key corresponding to the public key of at least one user.

30. The system of claim 25 wherein the means for encrypting the file encryption key includes means for accessing a public key of at least one recovery agent.

31. The system of claim 28 wherein the means for decrypting the file encryption key includes means for receiving a private key corresponding to the public key of at least one recovery agent.

32. The system of claim 24 wherein the encryption mechanism includes an installable software encryption driver.

33. The system of claim 32 wherein the encryption mechanism further includes a library of functions called by the encryption driver through the file system.

34. The system of claim 33 further comprising a generic data transformation mechanism for communicating between the encryption driver and the library of functions.

35. The system of claim 25 wherein the encryption mechanism and the decryption mechanism include a common installable software encryption driver.

36. The system of claim 25 further comprising means for verifying the integrity of the retrieved file encryption key.

37. The system of claim 36 wherein the means for verifying includes means for storing with the file encrypted information based on the file encryption key.

38. The system of claim 25 wherein the encryption mechanism includes an installable software encryption driver and the means for obtaining a file encryption key includes an interface to a cryptography service, and further comprising means for encrypting communications between the service and the encryption driver.

39. The system of claim 24 wherein the means for obtaining the file encryption key includes a key context buffer accessible at the file system software level for temporarily storing file encryption key information in association with the file.

40. The system of claim 25 wherein the means for retrieving the file encryption key includes a key context buffer accessible at the file system software level for temporarily storing file encryption key information in association with the file.

41. The method of claim 1 wherein writing encryption key information in association with the file includes writing file metadata.

42. The method of claim 1 wherein writing encryption key information in association with the file includes writing information about an encryption algorithm.

43. A computer-readable medium having computer-executable instructions for performing the method of claim 1.

44. The method of claim 14 wherein obtaining a file encryption key includes reading file metadata.

45. The method of claim 14 further comprising reading information about an encryption algorithm from the file encryption key data.

46. A computer-readable medium having computer-executable instructions for performing the method of claim 14.

47. The method of claim 22 wherein writing the encrypted file encryption key to the same non-volatile storage as the encrypted file data and in association therewith includes writing file metadata.

48. The method of claim 22 further comprising writing information about an encryption algorithm to the same non-volatile storage as the encrypted file data and in association therewith.

49. A computer-readable medium having computer-executable instructions for performing the method of claim 22.

50. In a computer system having a file system, a system for encrypting data written by the file system to a non-volatile storage, comprising, a file encryption key, a software encryption mechanism connected to the file system and configured to convert unencrypted data to encrypted data based on the file encryption key, the file system communicating with the encryption mechanism to write at least some of the data as encrypted data to a file in the non-volatile storage, a mechanism configured to encrypt the file encryp-

tion key into at least one set of encrypted key data, the file system writing each set of encrypted key data to the same non-volatile storage as the encrypted data and in association therewith.

51. The system of claim **50** wherein the file system writes each set of encrypted key data as metadata associated with the encrypted data.

52. The system of claim **50** wherein the system further includes a mechanism configured to obtain the encryption key from one of the sets of encrypted key data, and a decryption mechanism connected to the file system and configured to convert encrypted data read from the file into unencrypted data via the file encryption key.

53. The system of claim **50** wherein the mechanism configured to encrypt the file encryption key uses a plurality of public keys to encrypt the file encryption key into a plurality of sets of encrypted key data.

54. The system of claim **53** wherein at least one of the public keys corresponds to a user and at least one other of the public keys corresponds to a recovery agent.

55. The system of claim **50** further wherein at least one set of file encryption key data includes information about an encryption algorithm.

56. In a computer system having a file system, a method of returning requested file data, comprising:

receiving at file system software a request to read file data of an encrypted file;

determining whether file data corresponding to the request is stored on a storage medium or has been decrypted to an access-controlled location; and

if the file data has been decrypted to the access-controlled location, returning the file data in decrypted form from the access-controlled location in response to the request; or

if the file data is stored on the storage medium, reading the file data corresponding to the request from the storage medium, decrypting the file data at the file system software into unencrypted file data, and returning the unencrypted file data in response to the request.

57. The method of claim **56** wherein the access-controlled location comprises a cache.

58. The method of claim **56** wherein determining whether file data corresponding to the request is stored on the storage medium or has been decrypted to the access-controlled location comprises evaluating caching status data.

* * * * *